(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

WIPO

PCT

(43) International publication date
1 March 2001 (01.03.2001)

(10) International publication number
WO 01/14958 A2

(51) International patent classification[7]: G06F 9/00

(21) International application number: PCT/FR00/02349

(22) International filing date: 21 August 2000 (21.08.2000)

(25) Language of filing: French

(26) Language of publication: French

(30) Data relating to the priority:
99/10,697 23 August 1999 (23.08.1999) FR

(71) Applicant *(for all designated States except US)*:
TRUSTED LOGIC [FR/FR]; 23, avenue de Fulpmes, F-78450 Villepreux (FR).

(72) Inventor; and
(75) Inventor/Applicant *(US only)*: LEROY, Xavier [FR/FR]; 88 bis, avenue de Paris, F-78000 Versailles (FR).

(74) Representative: FRECHEDE, Michel etc.; Cabinet Plasseraud, 84, rue d'Amsterdam, F-75440 Paris Cedex 09 (FR).

(81) Designated states *(national)*: AU, CA, CN, JP, US.

(84) Designated states *(regional)*: European Patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE).

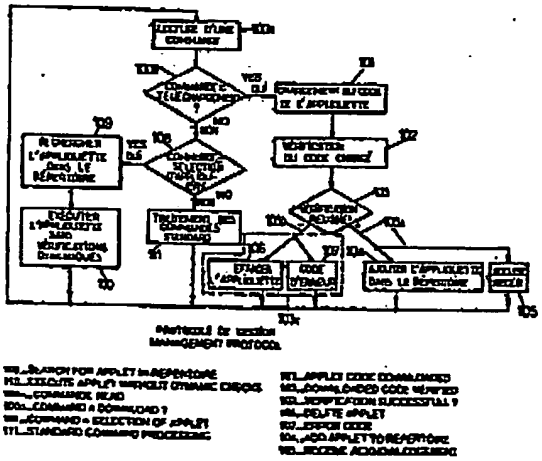
Published:
- *Without the International Search Report and to be republished once the report has been received.*

[continued on next page]

As printed

(54) Title: MANAGEMENT PROTOCOL, METHOD FOR VERIFYING AND TRANSFORMING A DOWNLOADED PROGRAMME FRAGMENT AND CORRESPONDING SYSTEMS

(54) Titre: PROTOCOLE DE GESTION, PROCEDE DE VERIFICATION ET DE TRANSFORMATION D'UN FRAGMENT DE PROGRAMME TELECHARGE ET SYSTEMES CORRESPONDANTS



(57) Abstract: The invention relates to a management protocol and to a method for verifying a programme fragment, or applet, which has been downloaded onto a portable system. An applet downloading command (100a, 100b) is executed. Once a positive response has been received, the object code of the applet is read (101) and subjected (102) to a verification process, instruction by instruction. The verification process consists of a stage comprising the initialisation of the type stack and table of register types representing the state of the virtual machine of the portable system at the start of the execution of the applet code; and a verification, instruction by instruction, for each target current instruction, of the existence of a target branch instruction, a target exception handler call or a target sub-routine call, the effect of the instruction on the type stack and the table of register types being verified and updated. If the verification is successful (103a), the applet is registered (104) and an acknowledgement is sent (105) to the downloading drive. Otherwise, the applet is destroyed (106). The invention is suitable for use for portable systems in a Java environment.

WO 01/14958 A2

For an explanation of the two-letter codes and the other abbreviations, reference is made to the explanations ("Guidance Notes on Codes and Abbreviations") at the beginning of each regular edition of the PCT Gazette.

(57) Abrégé: L'invention concerne un protocole de gestion et un procédé de vérification d'un fragment de programme, ou appliquette, téléchargé sur un système embarqué. Une commande de téléchargement (100a, 100b) de l'appliquette est effectuée. Sur réponse positive, le code objet de l'appliquette est lu (101) et soumis (102) à une vérification instruction par instruction. La vérification consiste en une étape d'initialisation de la pile des types et du tableau des types de registres représentant l'état de la machine virtuelle du système embarqué au début de l'exécution du code de l'appliquette et en une vérification, instruction par instruction pour chaque instruction courante cible, de l'existence d'une cible d'instruction de branchement, d'appel d'un gestionnaire d'exceptions ou d'un appel de sous-routine, et par une vérification et une actualisation de l'effet de cette instruction sur la pile des types et le tableau des types de registres. Sur vérification réussie (103a), l'appliquette est enregistrée (104) et un accusé de réception est envoyé (105) au lecteur de téléchargement. L'appliquette est détruite (106) sinon. Application aux systèmes embarqués en environnement Java.

PROTOCOL FOR MANAGING AND METHOD OF VERIFYING AND OF CONVERTING A DOWNLOADED PROGRAM FRAGMENT, AND CORRESPONDING SYSTEMS

The invention relates to a protocol for managing, a method of verifying and a method of transforming a downloaded program fragment and the corresponding systems, more particularly intended for on-board data-processing systems having few resources available in terms of memory and of computing power.

In a general way, by reference to figure 1a, it is reiterated that on-board data-processing systems 10 include a microprocessor 11, a permanent memory, such as a non-writable memory 12 containing the code of the executable program, and a rewritable, nonvolatile, permanent memory 15 of EEPROM type containing the data stored in the system, a volatile, random-access memory 14 in which the program stores its intermediate results while it is executing, and input/output devices 15 allowing the system to interact with its environment. In the case in which the on-board data-processing system consists of a microprocessor card, of the bank-card type, the input/output device 15 consists of a serial link allowing the card to communicate with a terminal, such as a card-reader terminal.

In conventional on-board data-processing systems, the code of the program executed by the system is fixed during construction of the system, or, at the latest, when the latter is customized before delivery to the final user.

More sophisticated on-board data-processing systems have, however, been implemented, these systems being reprogrammable, such as microprocessor cards of the JavaCard type, for example. With respect to the preceding types, these reprogrammable systems add the possibility of enhancing the program after the system has been put into service, via an operation of downloading program fragments. These fragments of programs, widely

designated by "applets", will be designated applets or program fragments indiscriminately in the present description. For a more detailed description of JavaCard systems, reference could usefully be made to the documentation published by the company SUN MICROSYSTEMS INC., and in particular to the electronically available documentation, JavaCard technology chapter, on the www (*World Wide Web*) site http://java.sun.com/products/javacard/index.html, June 1999.

figure 1b illustrates the architecture of such a reprogrammable on-board data-processing system. This architecture is similar to that of a conventional on-board system, with the difference that the reprogrammable on-board system can moreover receive applets by way of one of its input/output devices, then store them in its permanent memory 13 from which they can then be executed as a supplement to the main program.

For reasons of portability between different on-board data-processing systems, applets are presented in the form of code for a standard virtual machine. This code is not directly executable by the microprocessor 11, but it has to be interpreted in software terms by a virtual machine 16, which consists of a program resident in a non-writable permanent memory 12. In the abovementioned example of JavaCard cards, the virtual machine used is a subset of the Java virtual machine. For a description of the specifications relating to the Java virtual machine and of the virtual machine used, reference could usefully be made to the work published by Tim LINDHOLM and Frank YELLIN, entitled "The Java Virtual Machine Specification", Addison-Wesley 1996, and to the documentation published by the company SUN MICROSYSTEMS INC. "JavaCard 2.1 Virtual Machine Specification", documentation available electronically on the www site http://java.sun.com/products/javacard/JCVMSpec.pdf, March 1999.

The operation of downloading applets onto an on-board data-processing system in service poses considerable security problems. An applet which is involuntarily, or even deliberately, badly written may incorrectly modify data present on the system, prevent the main program from executing correctly or at the right time, or else modify other applets downloaded previously, making them unusable or harmful.

An applet written by a computer hacker may also divulge confidential information stored in the system, information such as the access code in the case of a bank card, for example. At the present time, three solutions have been proposed with a view to remedying the problem of security of applets.

A first solution consists in using cryptographic signatures, so as to accept only applets originating from trusted bodies or persons.

In the abovementioned example of a bank card, only the applets bearing the cryptographic signature of the bank having issued the card are accepted and executed by the card, any other unsigned applet being rejected in the course of the downloading operation. An ill-intentioned user of the card, not having available encryption keys from the bank, will therefore be incapable of executing an unsigned and dangerous applet on the card.

This first solution is well adapted to the case where all the applets originate from the same single source, the bank in the abovementioned example. This solution is difficult to apply in the case in which the applets originate from several sources, such as, in the example of a bank card, the manufacturer of the card, the bank, the bodies managing services by bank card, the large commercial distribution organizations offering clientele loyalty programs and proposing, legitimately, to download specific applets onto the card. The sharing and the holding among these various economic participants of the encryption keys necessary for the

electronic signature of the applets pose major technical, economic and legal problems.

A second solution consists in carrying out dynamic checks on access and on typing during the execution of the applets.

In this solution, the virtual machine carries out a certain number of checks, during the execution of the applets, such as:

- check of access to the memory: upon each read or write in a memory area, the virtual machine verifies the right of access by the applet to the corresponding data;
- dynamic verification of the data types: upon each instruction from the applet, the virtual machine verifies that the constraints on the data types are satisfied. By way of example, the virtual machine may have special handling for data such as valid memory addresses, and prevent the applet generating invalid memory addresses by way of integer/address conversions or arithmetic operations on the addresses;
- detection of stack overflows and of illegal accesses to the execution stack of the virtual machine, which, under certain conditions, are likely to disturb the operation thereof, to the point of circumventing the preceding check mechanisms.

This second solution allows execution of a wide range of applets under satisfactory security conditions. However, it features the drawback of a considerable slowing of the execution, caused by the range of dynamic verifications. In order to obtain a reduction in this slowing, some of these verifications can be taken charge of by the microprocessor itself, at the cost, however, of an increase in the complexity thereof and thus of the cost price of the on-board system. Such verifications furthermore increase the requirements for random-access and permanent memory of the system, by reason of the additional type information which it is necessary to associate with the data handled.

A third solution consists in carrying out a static verification of the code of the applet during the downloading.

In this solution, this static verification simulates the execution of the applet at the level of the data types and establishes, once and for all, that the code of the applet complies with the rule of data types and of access control imposed by the virtual machine and does not cause a stack overflow. If this static verification is successful, the applet can then be executed without it being necessary dynamically to verify that this rule is complied with. In the event that the static verification process fails, the on-board system rejects the "applet" and does not allow its subsequent execution. For a more detailed description of the abovementioned third solution, reference could usefully be made to the work published by Tim LINDHOLM and Frank YELLIN quoted above, to the article published by James A. GOSLING entitled "Java Intermediate Byte Codes", proceedings of the ACM SIGPLAN, Workshop on Intermediate Representations (IR'95), pages 111-118, January 1995, and to the US patent 5,748,964 granted on 05/05/1998.

Compared with the second solution, the third solution presents the advantage of a much more rapid execution of the applets, since the virtual machine does not carry out any verification during execution.

The third solution, however, features the drawback of a process of static verification of the code which is complex and expensive, both in terms of size of code necessary to conduct this process and in terms of size of random-access memory necessary to contain the intermediate results of the verification, and in terms of computation time. By way of illustrative example, the code verification integrated into the Java JDK system marketed by SUN MICROSYSTEMS represents about 50 kbytes of machine code, and its consumption in terms of random-access memory is proportional to (Tp + Tr) x Nb,

where Tp designates the maximum stack space, Tr designates the maximum number of registers and Nb designates the maximum number of branching targets used by a subprogram, also widely designated by method, of the applet. These memory requirements greatly exceeded the capacities of the resources of the majority of the present-day on-board data-processing systems, especially of commercially available microprocessor cards.

Several variants of the third solution have been proposed, in which the writer of the applet sends to the verifier, in addition to the code of the applet, a certain amount of specific supplementary information such as precalculated data types or preestablished proof of correct data typing. For a more detailed description of the corresponding operating modes, reference could usefully be made to the articles published by Eva ROSE and Kristoffer HØGSBRO ROSE, "Lightweight Bytecode Verification", proceedings of the Workshop Formal Underspinning of Java, October 1998, and by George C. NECULA, "Proof-Carrying Code", Proceedings of the 24th ACM Symposium Principles of Programming Languages, pages 106-119, respectively.

This supplementary information makes it possible to verify the code more rapidly and slightly to reduce the size of the code of the verification program but does not make it possible, however, to reduce the requirements for random-access memory, or even increases them, very substantially, in the case of the correct-data-typing preestablished-proof information.

The object of the present invention is to remedy the abovementioned drawbacks of the prior art.

In particular, one subject of the present invention is the implementation of a protocol for managing a downloaded program fragment, or applet, allowing execution of the latter by an on-board data-processing system having few resources available, such as a microprocessor card.

Another subject of the present invention is also the implementation of a method of verifying a downloaded program fragment, or applet, in which a process of static verification of the code of the applet is conducted when it is downloaded, this process possibly being aligned, at least in its principle, with the third solution described above, but in which new verification techniques are introduced, so as to allow execution of this verification within the limits of values of memory size and of computation speed imposed by the microprocessor cards and other low-power on-board data-processing systems.

Another subject of the present invention is also the implementation of methods of transforming program fragments of conventional type obtained, for example, by the use of a Java compiler on standardized program fragments, or applets, satisfying, a priori, verification criteria of the verification method which is the subject of the invention, with a view to accelerating the process of verifying and executing them at the level of present-day microprocessor cards or on-board data-processing systems.

Another subject of the present invention is, finally, the production of on-board data-processing systems allowing implementation of the abovementioned protocol for managing and of the abovementioned method of verifying a downloaded program fragment as well as of data-processing systems allowing implementation of the methods of transforming conventional program fragments, or applets, into standardized program fragments, or applets, as mentioned above.

The protocol for managing a downloaded program fragment on a reprogrammable on-board system, which is the subject of the present invention, applies especially to a microprocessor card equipped with a rewritable memory. The program fragment consists of an object code, a series of instructions, executable by the microprocessor of the on-board system by way of a vir-

tual machine equipped with an execution stack and with local variables or registers manipulated via these instructions and making it possible to interpret this object code. The on-board system is interconnected to a terminal.

It is noteworthy in that it consists at least, at the level of the on-board system, in detecting a command for downloading of the program fragment. On a positive response to the stage consisting in detecting a downloading command, it further consists in reading the object code constituting the program fragment and in temporarily storing this object code in the rewritable memory. The whole of the object code stored in memory is subjected to a verification process, instruction by instruction. The verification process consists at least in a stage of initializing the type stack and the table of register types representing the state of the virtual machine at the start of the execution of the temporarily stored object code and in a succession of stages of verification, instruction by instruction, of the existence, for each current instruction, of a target, branching-instruction target, target of an exception handler, and in a verification and an updating of the effect of the current instruction on the type stack and on the table of register types. In the event of an unsuccessful verification of the object code, the protocol which is the subject of the invention consists in deleting the momentarily recorded program fragment, when omitting to record the latter in the directory of available program fragments, and in sending an error code to the reader.

The method of verifying a program fragment downloaded onto an on-board system, which is the subject of the invention, applies especially to a microprocessor card equipped with a rewritable memory. The program fragment consists of an object code and includes at least one subprogram, a series of instructions, executable by the microprocessor of the on-board system by

way of a virtual machine equipped with an execution stack and with operand registers manipulated by these instructions, and making it possible to interpret this object code. The on-board system is interconnected to a reader.

It is noteworthy in that, following the detection of a downloading command and the storage of the object code constituting the program fragment in the rewritable memory, it consists, for each subprogram, in carrying out a stage of initializing the type stack and the table of register types by data representing the state of the virtual machine at the start of the execution of the temporarily stored object code, in carrying out a verification of the temporarily stored object code instruction by instruction, by discerning the existence, for each current instruction, of a branching-instruction target, of a target of an exception-handler call or of a target of a subroutine call, and in carrying out a verification and an updating of the effect of the current instruction on the data types of the type stack and of the table of register types, on the basis of the existence of a branching-instruction target, of a target subroutine call or of a target of an exception-handler call. The verification is successful when the table of register types is not modified in the course of a verification of all the instructions, the verification process being carried out instruction by instruction until the table of register types is stable, with no modification present. Otherwise the verification process is interrupted.

The method of transforming an object code of a program fragment into a standardized object code for this same program fragment, which is the subject of the present invention, applies to an object code of a program fragment in which the operands of each instruction belong to the data types manipulated by this instruction, the execution stack does not exhibit any overflow phenomenon and, for each branching instruction, the

type of the variables of the stack at this branching is the same as at the targets of this branching. The standardized object code obtained is such that the operands of each instruction belong to the data types manipulated by this instruction, the execution stack does not exhibit any overflow phenomenon and the execution stack is empty at each branching-target instruction.

It is noteworthy in that it consists, for all the instructions of the object code, in annotating each current instruction with the data type of the execution stack before and after the execution of this instruction, the annotation data being calculated by means of an analysis of the data stream relating to this instruction, in detecting, within the instructions and within each current instruction, the existence of branchings for which the execution stack is not empty, the detection operation being carried out on the basis of the annotation data of the type of stack variables allocated to each current instruction. In the presence of a detection of a non-empty execution stack, it further consists in inserting instructions to transfer stack variables on either side of these branchings or of these branching targets in order to empty the contents of the execution stack into temporary registers before this branching and to reestablish the execution stack from the temporary registers after this branching, and in not inserting any transfer instruction otherwise.

This method thus makes it possible to obtain a standardized object code for this same program fragment, in which the execution stack is empty at each branching instruction and branching-target instruction, in the absence of any modification to the execution of the program fragment.

The method of transforming an object code of a program fragment into a standardized object code for this same program fragment, which is the subject of the present invention, applies, moreover, to an object code

of a program fragment in which the operands of each instruction belong to the data types manipulated by this instruction, and a operand of given type written into a register by an instruction of this object code is reread from this same register by another instruction of this object code with the same given data type. The standardized object code obtained is such that the operands belong to the data types manipulated by this instruction, one and the same data type being allocated to the same register throughout the standardized object code.

It is noteworthy in that it consists, for all the instructions of the object code, in annotating each current instruction with the data type of the registers before and after the execution of this instruction, the annotation data being calculated by means of an analysis of the data stream relating to this instruction, and in carrying out a reallocation of the original registers employed with different types, by dividing these original registers into separate standardized registers. One standardized register is allocated to each data type used. Reupdating of the instructions which manipulate the operands which use the standardized registers is carried out.

The protocol for managing a program fragment, the method of verifying a program fragment, the methods of transforming object code of program fragments into standardized object code and the corresponding systems, which are the subjects of the present invention, find an application in the development of reprogrammable onboard systems, such as microprocessor cards, especially in the Java environment.

They will be better understood on reading the description and on perusing the drawings below, in which, other than figures 1a and 1b relating to the prior art:

- Figure 2 represents a flow chart illustrating the protocol for managing a program fragment downloaded onto a reprogrammable on-board system,

- Figure 3a represents, by way of illustration, a flow chart of a method of verifying a downloaded program fragment in accordance with the subject of the present invention,

- Figure 3b represents a diagram illustrating data types and sub-typing relationships implemented by the method of managing and the method of verifying a downloaded program fragment, which is the subject of the present invention,

- Figure 3c represents a detail of the verification method as claimed in figure 3a, relating to the managing of a branching instruction,

- Figure 3d represents a detail of the verification method as claimed in figure 3a, relating to the managing of a subroutine-call instruction,

- Figure 3e represents a detail of the verification method as claimed in figure 3a, relating to the managing of an exception-handler target,

- Figure 3f represents a detail of the verification method as claimed in figure 3a, relating to the managing of a target of incompatible branchings,

- Figure 3g represents a detail of the verification method as claimed in figure 3a, relating to the managing of an absence of branching target,

- Figure 3h represents a detail of the verification method as claimed in figure 3a, relating to the managing of the effect of the current instruction on the type stack,

- Figure 3i represents a detail of the verification method as claimed in figure 3a, relating to the managing of an instruction for reading a register,

- Figure 3j represents a detail of the verification method as claimed in figure 3a, relating to the managing of an instruction for writing to a register,

- Figure 4a represents a flow chart illustrating a method of transforming an object code of a program fragment into a standardized object code for this same program fragment with a branching instruction, respectively a branching-target instruction, with an empty stack,

- Figure 4b represents a flow chart illustrating a method of transforming an object code of a program fragment into a standardized object code for this same program fragment, making use of typed registers, a single specific data type being attributed to each register,

- Figure 5a represents a detail of implementation of the transformation method illustrated in figure 4a,

- Figure 5b represents a detail of implementation of the transformation method illustrated in figure 4b,

- Figure 6 represents a functional diagram of the complete architecture of a system for development of a standardized program fragment, and of a reprogrammable microprocessor card allowing implementation of the protocol for managing and the method of verifying a program fragment in accordance with the subject of the present invention.

In general, it is indicated that the protocol for managing and the method of verifying and transforming a downloaded program fragment, which is the subject of the present invention, and the corresponding systems, are implemented thanks to a software architecture for secure downloading and execution of applets on an on-board data-processing system with few resources, such as, in particular, microprocessor cards.

In general, it is indicated that the description below concerns the application of the invention in the context of reprogrammable microprocessor cards of JavaCard type, cf. documentation available electronically from the company SUN MICROSYSTEMS INC., JavaCard

Technology heading mentioned previously in the description.

However, the present invention is applicable to any on-board system which is reprogrammable by downloading an applet which is written in the code of a virtual machine including an execution stack, local variables or registers, and of which the execution model is strongly typed, each instruction of the code of the applet applying only to specific data types. The protocol for managing a program fragment downloaded onto a reprogrammable on-board system, which is the subject of the present invention, will now be described in more detail with reference to fig. 2.

With reference to the abovementioned figure, it is indicated that the object code which makes up the program fragment or applet consists of a series of instructions which can be executed by the microprocessor of the on-board system by means of the abovementioned virtual machine. The virtual machine makes it possible to interpret the abovementioned object code.
The on-board system is interconnected to a terminal via, for instance, a serial link.

With reference to the abovementioned fig. 2, the management protocol which is the subject of the present invention consists at least, in the on-board system, in detecting a command to download this program fragment in a stage 100a, 100b. Thus, stage 100a may consist of a stage of reading the abovementioned command, and stage 100b of a stage of testing the command which has been read and verifying the existence of a downloading command.

On a positive response to the abovementioned stage 100a, 100b of detecting a downloading command, the protocol which is the subject of the present invention subsequently consists in reading, at stage 101, the object code which makes up the relevant program fragment, and temporarily storing the abovementioned object code in the memory of the on-board data-

processing system. The abovementioned temporary storing operation can be executed either in the rewritable memory or, if appropriate, in the random-access memory of the on-board system, when this has sufficient capacity. The stage of reading the object code and temporarily storing it in the rewritable memory is designated as loading the code of the applet in fig. 2.

The abovementioned stage is then followed by a stage 102 consisting in submitting the whole of the temporarily stored object code to a process of verification, instruction by instruction, of the abovementioned object code.

The verification process consists, at least in a stage of initializing the stack of types and the table of data types representing the state of the virtual machine at the start of execution of the temporarily stored object code, and in a succession of stages of verifiying, instruction by instruction, by discerning the existence, for each current instruction, designated I_i, of a target such as a branching-instruction target designated CIB, a target of an exception-handler call or a target of a subroutine call. A verification and update of the effect of the current instruction I_i on the stack of types and on the table of register types is carried out.

When the verification has been successful at stage 103a, the protocol which is the subject of the present invention consists in recording, at stage 104, the downloaded program fragment in a directory of available program fragments, and in sending to the reader, at stage 105, a positive reception acknowledgment.

On the other hand, in the case of unsuccessful verification of the object code at stage 103b, the protocol which is the subject of the present invention consists in inhibiting, in a stage 103c, any execution on the on-board system of the momentarily recorded program fragment. The inhibition stage 103c can be imple-

mented in various ways. As a nonlimiting example, this stage can consist in deleting, at stage 106, the momentarily recorded program fragment, without recording this program fragment in the directory of available program fragments and, at stage 107, in sending an error code to the reader. Stages 107 and 105 can be implemented either sequentially after stages 106 and 104 respectively, or in multitasking operation with them.

With reference to the same fig. 2, on a negative response to the stage consisting in detecting a downloading command at stage 100b, the protocol which is the subject of the present invention consists in detecting, in a stage 108, a command to select an available program fragment from a directory of program fragments and, on a positive response to stage 108, having detected the selection of an available program fragment, in calling, at stage 109, this selected available program fragment in order to execute it. Stage 109 is then followed by a stage 110 of executing the called available program fragment by way of the virtual machine, with no dynamic verification of variable types, rights of access to the objects which are manipulated by the called available program fragment, or overflow of the execution stack when each instruction is executed.

In the case where a negative response is obtained at stage 108, this stage consisting in detecting a command to select a called available program fragment, the protocol which is the subject of the present invention consists in proceeding, in a stage 111, to process the standard commands of the on-board system.

Regarding the absence of dynamic verification of type or rights of access to objects of, for instance, JavaCard type, it is indicated that this absence of verification does not compromise the security of the card, because the code of the applet has necessarily successfully passed verification.

More specifically, it is indicated that the code verification which is carried out, as claimed in the method which is the subject of the present invention, on the microprocessor card or on-board data-processing system is more selective than the customary verification of codes for the virtual Java machine as described in the work entitled "The Java Virtual Machine Specification" which was mentioned previously in the description.

However, any code of the Java virtual machine which is correct as far as the traditional Java verifier is concerned can be transformed into an equivalent code which is capable of passing successfully the code verification which is carried out on the microprocessor card.

Whereas it is possible to imagine writing directly Java codes which satisfy the abovementioned verification criteria mentioned in the context of implementing the protocol which is the subject of the present invention, a noteworthy object of the latter is also the implementation of a method of automatic transformation of any standard Java code into a standardized code for the same program fragment, necessarily satisfying the verification criteria implemented as mentioned above. The method of transformation into standardized code, and the corresponding system, will be described in detail subsequently in the description.

A more detailed description of the method of verifying a program fragment, or applet, in accordance with the subject of the present invention, will now be given with reference to fig. 3a and the subsequent figures.

In general, it is indicated that the verification method which is the subject of the present invention can be implemented either as part of the protocol for managing a program fragment which is the subject of the present invention as described above with reference

to fig. 2, or independently, to provide whatever verification process is judged necessary.

In general, it is indicated that a program fragment is made up of an object code including at least one subprogram, more commonly designated a method, and is made up of a series of instructions which can be executed by the microprocessor of the on-board system via the virtual machine.

As shown in fig. 3a, the verification method consists, for each subprogram, in carrying out a stage 200 of initializing the stack of types and the table of register types of the virtual machine by data representing the state of this virtual machine at the start of execution of the object code which is the subject of verification. This object code can be stored temporarily as described above with reference to implementation of the protocol which is the subject of the present invention.

The abovementioned stage 200 is then followed by a stage 200a consisting in positioning the reading of the current instruction I_i, index i, on the first instruction of the object code. Stage 200a is followed by a stage 201 consisting in carrying out a verification of the abovementioned object code, instruction by instruction, by discerning the existence, for each current instruction, designated I_i, of a branching-instruction target CIB, of a target of an exception-handler call, designated CEM, or of a target of a subroutine call CSR.

The verification stage 201 is followed by a stage 202 of verifying and updating the effect of the current instruction I_i on the data types of the stack of types and of the table of register types, as a function of the existence, for the current instruction which is pointed at by another instruction, of a branching-instruction target CIB, of a target of a subroutine call CSR or a target of an exception-handler call CEM.

Stage 202 for the current instruction I_i is followed by a stage 203 to test whether the last instruction has been reached, the test written as:

I_i = last instruction of the object code?

On a negative response to test 203, the process passes to the next instruction 204, written i = i+1, and to the return to stage 201.

It is indicated that the abovementioned verification, at stage 202, has been successful when the table of register types is not modified during verification of all the instructions I_i which make up the object code. For this purpose, a test 205 of the existence of a stable state of the table of register types and provided. This test is written:

∃? Stable state of table of register types.

On a positive response to test 205, verification has been successful.

On the other hand, in the case where no absence of modification is noticed, the verification process is repeated and reinitiated by returning to stage 200a. It is demonstrated that the process is guaranteed to end after a maximum of NrxH iterations, where Nr designates the number of registers used and H designates a constant depending on the subtyping relation.

Various indications concerning the types of variables which are manipulated in the course of the verification process described above with reference to fig. 3a will now be given with reference to fig. 3b.

The abovementioned variable types include at least class identifiers corresponding to object classes which are defined in the program fragment which is subjected to verification, numeric variable types including at least a type short, an integer coded on p bits, where the value of p can be 16, and a type for the return address of a jump instruction JSR, this address type being identified as retaddr, a type null relating to references of null objects, a type object relating to objects proper, a specific type $\perp$ representing the

intersection of all the types and corresponding to the value zero nul, another specific type T representing the union of all the types and corresponding to any type of values.

With reference to fig. 3b, it is indicated that all the abovementioned variable types verify a subtyping relation:

object ε T;

short, retaddr ε T;

$\perp \varepsilon$ null, short, retaddr

A more specific example of a process of verification as illustrated in fig. 3a will now be given, with reference to a first example of a data structure, which is shown in table T1 in the annex.

The abovementioned example concerns an applet written in Java code.

The verification process accesses the code of the subprogram which forms the applet which is subjected to verification via a pointer to instruction I_i which is being verified.

The verification process records the size and type of the execution stack at the current instruction I_i corresponding to saload in the example of the abovementioned Table T1.

The verification process then records the size and type of the execution stack at the current instruction in the stack of types via its type stack pointer.

As mentioned above in the description, this stack of types reflects the state of the execution stack of the virtual machine at the current instruction I_i. In the example shown in table T1, at the time of the future execution of instruction I_i, the stack will contain three entries: a reference to an object of class C, a reference to a table of integers coded on p = 16 bits, the type short[], and an integer of p = 16 bits of type short. This is also shown in the type stack, which also contains three entries: C, the type of ob-

jects of class C, short[], the type of tables of integers p = 16 bits and short, the type of integers p = 16 bits.

Another noteworthy data structure consists of a table of register types, this table reflecting the state of the registers, that is to say of the registers which store the local variables, of the virtual machine.

Continuing the example indicated in table T1, it is indicated that entry 0 of the table of register types contains type C, i.e. at the time of the future execution of the current instruction I_i = saload, register 0 is guaranteed to contain a reference to an object of class C.

The various types which are manipulated during verification and stored in the table of register types and in the type stack are represented in fig. 3b. These types include:

- class identifiers CB corresponding to specific object classes which are defined in the applet;
- base types, such as short, an integer coded on p = 16 bits, int1 and int2, the most and least significant p bits respectively of integers coded on, e.g., 2p bits, or retaddr, the return address of an instruction as mentioned above;
- the type null, representing the references of null objects.

Regarding the subtyping relation, it is indicated that a type T1 is a subtype of a type T2 if every valid value of type T1 is also a valid value of type T2. The subtyping between class identifier reflects the inheritance hierarchy between classes of the applet. On the other types, subtyping is defined by the lattice shown in fig. 3b, where $\perp$ is a subtype of all types and all types are subtypes of T.

The sequence of the process of verifying a subprogram which forms an applet is as follows, referring to the abovementioned table T1.

The verification process is carried out independently on each subprogram of the applet. For each subprogram, the process carries out one or more verification passes on the instructions of the relevant subprogram. The pseudocode of the verification process is given in table T2 in the annex.

The process of verifying a subprogram begins with initializing the type stack and the table of register types shown in table T1, this initialization reflecting the state of the virtual machine at the start of execution of the subprogram being examined.

The type stack is initially empty, the stack pointer equals zero, and the register types are initialized with the types of the parameters of the subprogram, illustrating the fact that the virtual machine passes the parameters of this subprogram in these registers. The register types which are allocated by the subprogram are initialized to data types $\perp$, illustrating the fact that the virtual machine initializes these registers to zero at the start of execution of the subprogram.

Next, one or more verification passes on the instructions and on each current instruction I_i of the subprogram are carried out.

At the end of the implemented verification pass, or of a succession of passes for instance, the verification process determines whether the register types contained in the table of register types shown in table T1 of the annex have changed during the verification pass. In the absence of change, verification is terminated and a success code is returned to the main program, which makes it possible to send the positive reception acknowledgment at stage 105 of the management protocol shown in fig. 2.

If a change to the abovementioned table of register types is present, the verification process repeats the verification pass until the register types contained in the table of register types is stable.

The sequence proper of a verification pass which is carried out one or more times until the table of register types is stable will now be described with reference to figs. 3c to 3j.

For each current instruction I_i, the following verifications are carried out:

With reference to fig. 3a at stage 201, the verification process determines whether the current instruction I_i is the target of a branching instruction, a subroutine call or an exception-handler call, as mentioned above. This verification is carried out by examining the branching instructions in the code of the subprogram and the exception handlers associated with this subprogram.

With reference to fig. 3c which begins with stage 201, when the current instruction I_i is the target of a branching instruction, this condition being implemented by a test 300 designated by I_i = CIB, this branching being unconditional or conditional, the verification process checks that the type stack is empty at this point of the subprogram by a test 301. On a positive response to the test 301, the verification process is continued by a context continuation stage marked continue A. On a negative response to the test 301, the type stack not being empty, the verification fails and the applet is rejected. This failure is represented by the Failure stage.

With reference to fig. 3d which begins with the continue A stage, when the current instruction I_i is the target of a subroutine call, this condition being implemented by a test 304 I_i = CSR, the verification process verifies, in a test 305, that the previous instruction I_{i-1} does not continue in sequence. This verification is implemented by a test stage 305 when the previ-

ous instruction is an unconditional branching, a subroutine return or a raising of an exception. The test at stage 305 is marked as follows:

I_{i-1} = $IB_{unconditional}$, return RSR or raising L-EXCEPT.

On a negative response to test 305, the verification process fails in a Failure stage. On the other hand, on a positive response to test 305, the verification process reinitializes the type stack in such a way that it contains exactly one entry of retaddr type, the return address of the abovementioned subroutine. If the current instruction I_i at stage 304 is not the target of a subroutine call, the verification process is continued in the context at the continue B stage.

With reference to fig. 3e, when the current instruction I_i is the target of an exception handler, this condition being implemented by a test 307 marked I_i = CEM, where CEM designates the target of an exception handler, this condition is implemented by a test 307, marked:

I_i = CEM.

On a positive response to test 307, the process verifies that the previous instruction is an unconditional branching, a subroutine return or a raising of exceptions by a test 305, marked:

I_{i-1} = $IB_{unconditional}$, return RSR or raising L-EXCEPT.

On a positive response to test 305, the verification process proceeds to reupdate the type stack, at a stage 308, by entering exception types, marked EXCEPT type, stage 308 being followed by a context continuation stage, continue C. On a negative response to test 305, the verification process fails by the stage marked Failure. The program fragment is then rejected.

With reference to fig. 3f, when the current instruction I_i is the target of multiple incompatible branchings, this condition is implemented by a test 309, which is marked:

I_i = incompatible XIBs

incompatible branchings being, for instance, an unconditional branching and a subroutine call, or two different exception handlers. On a positive response to test 309, the branchings being incompatible, the verification process fails by a stage marked Failure and the program fragment is rejected. On a negative response to test 309, the verification process is continued by a stage marked continue D. Test 309 begins with the continue C stage which was mentioned previously in the description.

With reference to fig. 3g, when the current instruction I_i is not the target of any branching, this condition being implemented by a test 310 beginning with the abovementioned continue D, this test being marked

I_i ∃? branching targets,

where ∃ designates the existence symbol,

the verification process continues on a negative response to the test 310 by passing to an update of the type stack at stage 311, stage 311 and the positive response to test 310 being followed by a context continuation stage at stage 202, which is described above in the description with reference to fig. 3a.

A more detailed description of the stage of verifying the effect of the current instruction on the type stack at the abovementioned stage 202 will now be given with reference to fig. 3h.

According to the abovementioned figure, this stage can include at least one stage 400 of verification that the type execution stack includes at least as many entries as the current instruction includes operands. This test stage 400 is marked:

$Nbep \geq NOpi$

where Nbep designates the number of entries of the type stack and NOpi designates the number of operands included in the current instruction.

On a positive response to test 400, this test is followed by a stage 401a of unstacking the type stack, and of verifying 401b that the types of the entries at the top of the stack are subtypes of the types of the operands of the abovementioned current instruction. At test stage 401a, the operand types of the instruction i are marked TOpi, and the types of the entries at the top of the stack are marked Targs.

At stage 401b, the verification corresponds to a verification of the subtyping relation Targs subtype of TOpi.

On a negative response to tests 400 and 401b, the verification process fails, which is shown by access to the Failure stage. On the other hand, on a positive response to test 401b, the verification process is continued, and consists in carrying out:

- A stage of verifying the existence of a sufficient memory space on the type stack to proceed to stack the results of the current instruction. This verification stage is implemented by a test 402, marked:

Stack-space $\geq$ Results-space

where each side of the inequality designates the corresponding memory space.

On a negative response to test 402, the verification process fails, which is shown by the Failure stage. On the other hand, on a positive response to test 402, the verification process then proceeds to stack the data types which are assigned to the results in a stage 403, the stacking being done on the stack of data types which are assigned to these results.

As a nonlimiting example, it is indicated that to implement fig. 3h for verifying the effect of the current instruction on the type stack, for a current instruction consisting of a Java saload instruction corresponding to reading an integer element coded on p = 16 bits in a table of integers, this table of integers being defined by the table of integers and an in-

teger index in this table, and the result by the integer which is read at this index in this table, the verification process checks that the type stack contains at least two elements, that the two elements at the top of the type stack are subtypes of short[] and short respectively, proceeds to the unstacking process and then to the process of stacking the data type short as the result type.

Additionally, with reference to fig. 3i, to implement the stage of verifying the effect of the current instruction on the type stack, when the current instruction I_i is a read instruction, marked IR, of a register of address n, this condition being implemented by a test 404 marked $I_i = IR_n$, on a positive response to the abovementioned test 404, the verification process consists in verifying the data type of the result of this reading, in a stage 405, by reading the entry n in the table of register types, then in determining the effect of the current instruction I_i on the type stack by an operation 406a of unstacking the entries of the stack corresponding to the operands of this current instruction and by stacking 406b the data type of this result. The operands of the instruction I_i are marked OP_i. Stages 406a and 406b are followed by a return to the context continuation, continue F. On a negative response to test 404, the verification process is continued by the context continuation, continue F.

With reference to fig. 3j, when the current instruction I_i is a write instruction, marked IW, of a register of address n, this condition being implemented by a test marked $I_i = IW_m$, on a positive response to test 407, the verification process consists in determining, in a stage 408, the effect of the current instruction on the type stack and the type t of the operand which is written in the register of address n, then, in a stage 409, in replacing the type entry of the table of register types at address n by the type immediately above the previously stored type and above

the type t of the operand which is written in the register of address n. Stage 409 is followed by a return to the context continuation, continue 204. On a negative response to test 407, the verification process is continued by a context continuation, continue 204.

As an example, when the current instruction I_i corresponds to writing a value of type D into a register of address 1, and the type of register 1 before verification of the instruction was C, the type of register 1 is replaced by the type object, which is the smallest type which is higher than C and D in the lattice of types shown in fig. 3b.

In the same way, as an example, when the current instruction I_i is a read of an instruction aload-0 consisting in stacking the contents of register 0, and entry 0 of the table of register types is C, the verifier stacks C onto the type stack.

An example of verifying a subprogram written in a Java environment will now be given, with reference to tables T3 and T4 in the annex.

Table T3 represents a specific JavaCard code corresponding to the Java subprogram which is included in this table.

table T4 shows the contents of the table of register types and of the type stack before verification of each instruction. The type constraints on the operands of the various instructions are all observed. The stack is empty both after the instruction 5 to branch to instruction 9, symbolized by the arrow, and before the abovementioned branching target 9. The type of register 1, which was initially $\perp$, becomes null, the upper bound of null and $\perp$, when instruction 1 to store a value of type null in register 1 is examined, then becomes of type short[], the upper bound of types short[] and null, when instruction 8 to store a value of type short[] in register 1 is processed. The type of register 1 having changed during the first verification

pass, a second pass is carried out, distributing register types which were obtained at the end of the first. This second verification pass is successful, just like the first, and does not change the register types. The verification process thus terminates successfully.

Various examples of cases of failure of the verification process on four examples of incorrect code will now be given with reference to table T5 in the annex:

- At point a) of table T5, the purpose of the code given as an example is to attempt to fabricate an invalid object reference using an arithmetic process on pointers. It is rejected by verification of the types of arguments of instruction 2 sadd, which requires these two arguments to be of type short.
- At points b) and c) of table T5, the purpose of the code is to carry out two attempts to transform any integer into an object reference. At point b), register 0 is used simultaneously with type short, instruction 0, and with type null, instruction 5. Consequently, the verification process assigns type T to record 0, and detects a type error when register 0 is returned as a result of type object at instruction 7.
- At point c) of table T5, a set of branchings of type "if ... then ... else ..." is used to leave at the top of the stack a result which consists of either an integer or an object reference. The verification process rejects this code because it detects that the stack is not empty at the branching from instruction 5 to instruction 9, symbolized by the arrow.
- Finally, at point d) of table 5, the code contains a loop which, at each iteration, has the effect of stacking an additional integer on the top of the stack, and thus causing a stack overflow after a certain number of iterations. The

verification process rejects this code, noticing that the stack is not empty at the backward branching from instruction 8 to instruction 0, symbolized by the return arrow, the stack not being empty at a branching point.

The various examples given above with reference to tables T3, T4 and T5 show that the verification process, which is the subject of the present invention, is particularly efficient, and that it applies to applets, and in particular to subprograms thereof, for which the conditions of stack type, or respectively of the empty character of the type stack previously, and to the branching instructions or branching targets, are satisfied.

Obviously, such a verification process implies writing object codes which satisfy these criteria, these object codes possibly corresponding to the subprogram in the abovementioned table T3.

However, and in order to ensure verification of existing applets and subprograms of applets which do not necessarily satisfy the verification criteria of the method which is the subject of the present invention, in particular regarding applets and subprograms which are written in the Java environment, the purpose of the present invention is to establish methods of transforming these applets or subprograms into standardized applets or program fragments, making it possible to undergo successfully the verification tests of the verification method which is the subject of the present invention and of the management protocol which implements such a method.

For this purpose, the subject of the invention is the implementation of a method and a program for transforming a traditional object code forming an applet, it being possible to implement this method and this transformation program, outside an on-board system or microprocessor card, when the relevant applet is created.

The method of transforming code into standardized code, which is the subject of the present invention, will now be described in the framework of the Java environment, as a purely illustrative example.

The JVM codes which are produced by existing Java compilers satisfy various criteria, which are stated below:

C1: the arguments of each instruction actually belong to the types which this instruction expects;

C2: the stack does not overflow;

C'3: for each branching instruction, the type of the stack at this branching is the same as at the possible targets for this branching;

C'4: a value of type t which is written into a register at one point of the code and reread from the same register at another point of the code is always reread with the same type t;

The implementation of the verification method which is the subject of the present invention implies that criteria C'3 and C'4, which are verified by the object code which is submitted for verification, be replaced by criteria C3 and C4 below:

C3: the stack is empty at each branching instruction and at each branching target;

C4: the same register is used with one and the same type throughout the code of a subprogram.

With reference to the abovementioned criteria, it is indicated that Java compilers guarantee only the weaker criteria C'3 and C'4. The verification process which is the subject of the present invention and the corresponding management protocol in fact guarantee more restrictive criteria C3 and C4, making it possible to ensure the security of execution and management of applets.

The concept of standardization, covering the transformation of codes into standardized codes, can present various aspects, to the extent that replacement

of criteria C'3 and C'4 by criteria C3 and C4, in conformity with the verification process which is the subject of the present invention, can be implemented independently, to ensure that the stack is empty at each branching instruction and at each branching target, and respectively that the registers which the applet opens are typed, and a single data type which is assigned for execution of the relevant applet corresponds to each open register, or, on the other hand, jointly, to satisfy the whole of the verification process which is the subject of the present invention.

The method of transforming an object code into standardized object code as claimed in the invention will consequently be described as claimed in two distinct implementation modes, a first implementation mode corresponding to transformation of an object code which satisfies criteria C1, C2, C'3, C'4 into a standardized object code which satisfies criteria C1, C2, C3, C'4 corresponding to a standardized code with an empty branching instruction or branching target, then, as claimed in a second implementation mode, in which the traditional object code which satisfies the same initial criteria is transformed into a standardized object code which satisfies criteria C1, C2, C'3, C4, for instance corresponding to a standardized code using typed registers.

The first implementation mode of the code transformation method which is the subject of the present invention will now be described with reference to fig. 4a. In the implementation mode which is shown in fig. 4a, the initial traditional code is considered to satisfy criteria C1+C2+C'3, and the standardized code which is obtained as the result of the transformation is considered to satisfy criteria C1+C2+C3.

According to the abovementioned figure, the transformation method consists, for each current instruction I_i of the code or of the subprogram, in annotating each instruction, in a stage 500, with the data

type of the stack before and after execution of this instruction. The annotation data is marked AI_i and is associated by the relation $I_i \Leftrightarrow AI_i$ with the relevant current instruction. The annotation data is calculated by means of an analysis of the data stream relating to this instruction. The data types before and after execution of the instruction are marked tbe_i and tae_i respectively. Calculation of annotation data by analysis of the data stream is a traditional calculation which is known to those skilled in the art, and will therefore not be described in detail.

The operation which is implemented at stage 500 is illustrated in table T6 in the annex, in which, for an applet or subprogram of an applet including 12 instructions, the annotation data AI_i made up of the types of registers and the types of the stack are introduced.

The abovementioned stage 500 is then followed by a stage 500a consisting in positioning the index i on the first instruction $I_i = I_1$. Stage 500a is followed by a stage 501 consisting in detecting, among the instructions and in each current instruction I_i, the existence of branchings marked IB or of branching targets CIB for which the execution stack is not empty. This detection 501 is implemented by a test which is carried out on the basis of the annotation data AI_i of the type of stack variables allocated to each current instruction, the test being marked for the current instruction:

I_i is an IB or CIB and stack (AI) ? empty.

On a positive response to test 501, i.e. in the presence of detection of a non-empty execution stack, the abovementioned test is followed by a stage consisting in inserting instructions to transfer stack variables on either side of these branchings IB or branching targets CIB, in order to empty the content of the execution stack into temporary registers before this branching and to reestablish the execution stack from the temporary registers after this branching. The insertion stage is marked 502 in fig. 4a. It is followed

by a stage 503 to test reaching the last instruction, marked

I_i = last instruction?

On a negative response to the test 503, an increment 504 i=i+1 is carried out, to go on to the next instruction and return to stage 501. On a positive response to test 503, an End stage is initiated. On a negative response to test 501, the transformation method is continued by a branching to stage 503 in the absence of insertion of a transfer instruction. Implementation of the method of transforming a traditional code into a standardized code with branching instruction with empty stack as represented in fig. 4a makes it possible to obtain a standardized object code for the same initial program fragment in which the stack of stack variables is empty at each branching instruction and each branching-target instruction, in the absence of any modification to the execution of the program fragment. In the case of a Java environment, the instructions to transfer data between stack and register are the load and store instructions of the Java virtual machine.

Returning to the example introduced in table T6, the transformation method detects a branching target where the stack is not empty at instruction 9. The method is then to insert an instruction istore 1 before the branching instruction 5 which leads to the above-mentioned instruction 9, in order to save the content of the stack in register 1 and ensure that the stack is empty at the time of the branching. Symmetrically, an instruction iload 1 is inserted before the instruction target 9, to reestablish the content of the stack exactly as it was before the branching. Finally, an instruction istore 1 is inserted after instruction 8 to ensure that the stack is balanced on the two paths which lead to instruction 9. The result of the transformation carried out in this way into a standardized code is shown in table T7.

The second implementation mode of the transformation method which is the subject of the present invention will now be described with reference to fig. 4b in the case in which the initial traditional object code satisfies criteria C1+C'4 and the standardized object code satisfies criteria C1+C4.

With reference to the abovementioned fig. 4b, it is indicated that the method, in this implementation mode, consists in annotating, as claimed in a stage 500 which is approximately the same as that which is shown in fig. 4a, each current instruction I_i with the type of register data before and after execution of this instruction. In the same way, the annotation data AI_i is calculated by means of an analysis of the data stream relating to this instruction.

The annotation stage 500 is then followed by a stage consisting in carrying out a reallocation of the registers, the stage marked 601, by detecting the original registers employed with different types, by dividing these original registers into separate standardized registers, one standardized register being allocated to each data type used. Stage 601 is followed by a stage 602 of reupdating the instructions which manipulate the operands which use the abovementioned standardized registers. Stage 602 is followed by a context continuation stage 302.

With reference to the example given in table T6, it is indicated that the transformation method detects that the register of rank 0, marked r0, is used with the two types, object, instructions 0 and 1, and int, instruction 9 and following. The method is then to divide the original register r0 into two registers, register 0 for the use of object types and register 1 for uses of int type. References to record 0 of int type are then rewritten by transforming them into references to record 1, the standardized code obtained being shown in table T8 in the annex.

It is noted, in a nonlimiting way, that in the example which is introduced with reference to the abovementioned table T8, the new register 1 is used simultaneously for standardization of the stack and for creation of typed registers by dividing of register 0 into two registers.

The method of transforming a traditional code into a standardized code with branching instruction with empty stack as described in fig. 4a will now be described in more detail in a preferred, nonlimiting implementation mode, in relation to fig. 5a.

This implementation mode concerns stage 501, consisting in detecting, within the instructions and within each current instruction I_i, the existence of branching IB, or respectively of branching target CIB, for which the stack is not empty.

Following the determination of target instructions where the stack is not empty, this condition being marked at stage 504a, I_i stack≠empty, the transformation process consists in associating with these instructions, at the abovementioned stage 504a, a set of new registers, one per stack location which is active at these instructions. Thus, if i designates the rank of a branching target of which the associated stack type is not empty and is of type tpl_i to tpn_i with n > 0, stack not empty, the transformation process allocates n new, as yet unused, registers, r_1 to r_n, and associates them with the corresponding instruction i. This operation is implemented at stage 504a.

Stage 504a is followed by a stage 504 consisting in examining each detected instruction of rank i and identifying, in a test stage 504, the existence of a branching target CIB or of a branching IB. Stage 504 is shown in the form of a test designated by:

∃?CIB,IB and I_i=CIB.

In the case that the instruction of rank i is a branching target CIB represented by the preceding

equality, and that the stack of stack variables at this instruction is not empty, i.e. with a positive response to test 504, for every preceding instruction of rank i-1 consisting of a branching, a raising of an exception or a program return, this condition is implemented at test stage 505, designated by:

I_{i-1} = IB, EXCEPT raising, Prog. return.

The detected instruction of rank i is only accessible by a branching. On a positive response to the abovementioned test 505, the transformation process consists in carrying out a stage 506 consisting in inserting a set of load instructions of load type from the set of new registers before the relevant detected instruction of rank i. The insertion operation 506 is followed by a redirection 507 of all branchings to the detected instruction of rank i, to the first inserted load instruction load. The insertion and redirection operations are shown in table T9 in the annex.

For every preceding instruction of rank i-1 continuing in sequence, i.e. when the current instruction of rank i is accessible simultaneously by a branching and from the preceding instruction, this condition being implemented by test 508 and symbolized by the relations:

$I_{i-1} \rightarrow I_i$

and

$IB \rightarrow I_i$

the transformation process consists in a stage 509 of inserting a set of backup instructions store to back up to the set of new registers before the detected instruction of rank i, and a set of load instructions load to load from this set of new registers. Stage 509 is then followed by a stage 510 of redirection of all the branchings to the detected instruction of rank i to the first inserted load instruction load.

In the case that the detected instruction of rank i is a branching to a determined instruction, for

any detected instruction of rank i consisting of an unconditional branching, this condition being implemented by a test 511 marked:

$I_i = IB_{uncondit.}$

the transformation process as shown in fig. 5a consists in inserting at a stage 512, on a positive response to test 511, before the detected instruction of rank i, multiple backup instructions store. The transformation process inserts before the instruction i the n instructions store as shown in table T11 as an example. The instructions store address registers r_1 to r_n, where n designates the number of registers. Thus the backup instruction is associated with each new register.

For every detected instruction of rank i consisting of a conditional branching, and for a number mOp, greater than 0, of operands manipulated by this conditional branching instruction, this condition being implemented by the test 513 marked:

$I_i = IB_{condit.}$

with mOp > 0

the transformation process, in positive response to the abovementioned test 513, consists of inserting, at a stage 514 before this detected instruction of rank i, a permutation instruction marked swap_x at the top of the stack of stack variables of the mOp operands of the detected instruction of rank i and the n following values. This permutation operation makes it possible to collect at the top of the stack of stack variables the n values to be backed up in the set of new registers r_1 to r_n. Stage 514 is followed by a stage 515 consisting in inserting, before the instruction of rank i, a set of backup instructions store to back up to the set of new registers r_1 to r_n. The abovementioned insertion stage 515 is itself followed by a stage 516 of insertion, after the detected instruction of rank i, of a set of load instructions load to load from the set of new registers r_1 to r_n. The set of corresponding insertion operations is shown in table 12 in the annex.

For reasons of completeness and with reference to fig. 5a, it is indicated that, on a negative response to test 504, the continuation of the transformation process is implemented by a context continuation stage, continue 503, that the negative response to tests 505, 508, 511 and 513 is itself followed by a continuation of the transformation process via a context continuation stage, continue 503, and that the same applies to the continuation of operations after the abovementioned redirection stages 507 and 510 and insertion stages 512 and 516.

A more detailed description of the method of standardizing and transforming an object code into a standardized object code using typed registers as described in fig. 4b will now be given with reference to fig. 5b. This implementation mode concerns, more particularly, a nonlimiting, preferred implementation mode of stage 601 to reallocate registers by detecting the original registers used with different types.

With reference to the abovementioned fig. 5b, it is indicated that the abovementioned stage 601 consists in determining, in a stage 603, the lifetime intervals marked ID_j of each register r_j. These lifetime intervals, designated "*live range*" or "*webs*", are defined for a register r as a maximum set of partial traces such that register r is live at all points of these traces. For a more detailed definition of these concepts, it is useful to refer to the work edited by Steven S. MUCHNICK entitled "Advanced Compiler Design and Implementation", Section 16.3, Morgan KAUFMANN, 1997. Stage 603 is designated by the relation:

$$ID_j \longleftrightarrow r_j$$

as claimed in which a corresponding lifetime interval ID_j is associated with each register r_j.

The abovementioned stage 603 is followed by a stage 604 consisting in determining, at stage 604, the main data type, marked tp_j, of each lifetime interval

ID_j. The main type of a lifetime interval ID_j, for a register r_j, is defined by the upper bound of the data types stored in this register r_j by the backup instructions store belonging to the abovementioned lifetime interval.

Stage 604 is itself followed by a stage 605 consisting in establishing an interference graph between the lifetime intervals as defined above at stages 603 and 604, this interference graph consisting of a non-oriented graph of which each peak consists of a lifetime interval, and of which the arcs, marked $a_{j1,j2}$ on fig. 5b, between two peaks ID_{j1} and ID_{J2}, exist if a peak contains a backup instruction addressed to the register of the other peak or vice versa. In fig. 5b, the construction of the interference graph is shown symbolically, it being possible to implement this construction on the basis of calculation techniques which are known to those skilled in the art. For a more detailed description of the construction of this type of graph, it is useful to refer to the work published by Alfred V. AHO, Ravi SETHI and Jeffrey D. ULLMAN entitled "Compilers: principles, techniques, and tools", Addison-Wesley 1986, Section 9.7.

Following stage 605, the standardization method as shown in fig. 5b consists in translating, in a stage 606, the uniqueness of a data type which is allocated to each register r_j in the interference graph, by adding arcs between all pairs of peaks of the interference graph while two peaks of a pair of peaks do not have the same associated main data type. It is understood that the translation of the character of uniqueness of a data type which is allocated to each register obviously corresponds to translating and taking into account criterion C4 in the interference graph, this criterion being mentioned previously in the description. The abovementioned stage 606 is then followed by a stage 607 in which an instantiation of the interference graph is carried out, this instantiation being more

commonly designated as the painting stage of the interference graph as claimed in the usual techniques. During stage 607, the transformation process assigns to each lifetime interval ID_{jk} a register number rk, in such a way that two adjacent intervals in the interference graph receive different register numbers.

This operation can be implemented on the basis of any suitable process. As a nonlimiting example, it is indicated that a preferred process can consist:

a) in choosing a peak of minimum degree in the interference graph, minimum degree being defined as a minimum number of adjacent peaks, and withdrawing it from the graph. This stage can be repeated until the graph is empty.

b) Each previously withdrawn peak is reintroduced into the interference graph in the inverse order of their withdrawal, the last to be removed being the first to be reintroduced, and successively in the inverse order of the order of withdrawal. Thus the smallest register number which is different from the numbers assigned to all the adjacent peaks can be assigned to each reintroduced peak.

Finally, by stage 602, shown in fig. 4b, the transformation and reallocation process rewrites the access instructions to the registers in the code of the subprogram of the relevant applet. Access to a given register in the corresponding lifetime interval is replaced by access to a different register, the number of which has been assigned during the instantiation phase, also designated the painting phase.

A more detailed description of an on-board data-processing system, making it possible to implement the management protocol and verification process of a program fragment or applet as claimed in the subject of the present invention, and of a development system of an applet, will now be given with reference to fig. 6.

Regarding the corresponding on-board system with reference 10, it is recalled that this on-board system is a reprogrammable-type system, including the essential components as shown in fig. 1b. The abovementioned on-board system is considered to be interconnected to a terminal by a serial link, the terminal itself being linked, for instance via a local network, if appropriate a remote network, to an applet development computer with reference 20. On the on-board system 10 runs a main program which reads and executes the commands which the terminal sends on the serial link. Additionally, the standard commands for a microprocessor card, such as for instance the standard commands of the ISO 7816 protocol, can be implemented, and the main program recognizes two additional commands, one for remote loading of an applet, and the other for selecting an applet which has previously been loaded onto the microprocessor card.

In conformity with the subject of the present invention, the structure of the main program is implemented in such a way as to include at least one program module for management and verification of a downloaded program fragment, following the protocol for managing a downloaded program fragment as described above in the description with reference to fig. 2.

Additionally, the program module also includes a subprogram module to verify a downloaded program fragment, following the verification method as described above in the description with reference to figs. 3a to 3j.

For this purpose, the structure of the memories, in particular the non-writable permanent memory ROM, is modified in such a way as to include in particular, apart from the main program, a protocol management and verification module 17, as mentioned above. Finally, regarding the nonvolatile rewritable memory of EEPROM type, this advantageously includes a directory of applets, marked 18, making it possible to implement the

management protocol and verification process which are the subjects of the present invention.

With reference to the same fig. 6, it is indicated that the applet development system conforming to the subject of the present invention, in fact making it possible to transform a traditional object code as mentioned above in the description, and satisfying criteria C1+C2+C'3+C'4 in the framework of the Java environment, into a standardized object code for the same program fragment, includes, associated with a traditional Java compiler 21, a code transformation module, marked 22, which proceeds to transform code into standardized code as claimed in the first and second implementation modes described above in the description with reference to figs. 4a, 4b and 5a, 5b. It is in fact understood that, on the one hand, standardization of the original object code into a standardized object code with branching instruction with empty stack and into a standardized code using typed registers, on the other hand, as mentioned previously in the description, makes it possible to satisfy verification criteria C3 and C4, which are imposed by the verification method which is the subject of the present invention.

The code transformation module 22 is followed by a JavaCard transformer 23, which makes it possible to ensure transmission by a remote or local network to the terminal and, via the serial link, to the microprocessor card 10. Thus the applet development system 20 shown in fig. 6 makes it possible to transform the compiled class files produced by the Java compiler 21 from the Java source codes of the applet into class files which are equivalent, but which observe the additional constraints C3, C4 which are imposed by the management protocol and the verification module 17 on board the microprocessor card 10. These transformed class files are transformed into a downloadable applet on the card by the standard JavaCard transformer 23.

Various particularly noteworthy components of the set of protocol components, methods and systems which are the subjects of the present invention will now be given for information only.

Compared to the verification processes of the prior art as mentioned in the introduction to the description, the verification method which is the subject of the present invention appears noteworthy in that it concentrates the verification effort on the typing properties of the operands which are essential to the security of execution of each applet, i.e. observing the type constraints associated with each instruction and absence of stack overflow. Other verifications do not appear to be essential in terms of security, in particular verification that the code correctly initializes every register before reading it for the first time. On the contrary, the verification method which is the subject of the present invention operates by initializing to zero all the registers from the virtual machine when the method is initialized, to guarantee that reading a non-initialized register cannot compromise the security of the card.

Additionally, the demand imposed by the verification method which is the subject of the present invention, as claimed in which the stack must be empty at each branching or branching target instruction, guarantees that the stack is in the same state, empty, after execution of the branching and before execution of the instruction to which the program has branched. This mode of operation guarantees that the stack is in a consistent state, whatever the execution route which is followed through the code of the relevant subprogram or applet. The consistency of the stack is thus guaranteed even in the presence of a branching or branching target. Contrary to the methods and systems of the prior art, in which it is necessary to conserve in random-access memory the type of the stack at each branching target, which necessitates a quantity of random-access

memory proportional to TpxNb, the product of the maximum size of execution stack which is used and the number of branching targets in the code, the verification method which is the subject of the present invention only needs the type of the execution stack at the time of the instruction during verification, and it does not keep in memory the type of this stack at other points of the code. Consequently, the method which is the subject of the invention is satisfied with a quantity of random-access memory proportional to Tp but independent of Nb, and consequently of the length of the code of the subprogram or applet.

The requirement as claimed in criterion C4, as claimed in which a given register must be used with one and the same type throughout the code of a subprogram, guarantees that the abovementioned code does not use a register in an inconsistent way, e.g. by writing a short integer to it at one point of the program and rereading it as an object reference at another point of the program.

In the verification processes which are described in the prior art, in particular in the previously mentioned Java specification entitled "The Java Virtual Machine Specification", edited by Tim LINDHOLM and Frank YELLIN, to guarantee the consistency of the abovementioned uses through the branching instructions, it is necessary to keep in random-access memory a copy of the table of register types at each branching target. This operation necessitates a quantity of random-access memory proportional to $T_r \times N_b$, where T_r designates the number of registers used by the subprogram and N_b the number of branching targets in the code of this subprogram.

On the contrary, the verification process which is the subject of the present invention operates on a global table of register types without keeping a copy at different points of the code in random-access mem-

ory. Consequently, the random-access memory which is required to implement the verification process is proportional to T_r but independent of N_b, and consequently of the length of the code of the relevant subprogram.

The constraint as claimed in which a given register is used with the same type at all points, i.e. at every instruction of the relevant code, simplifies appreciably and significantly the verification of subprograms. On the contrary, in the verification processes of the prior art, in the absence of such a constraint, the verification process must establish that the subprograms observe a strict stack discipline, and must verify the body of the subprograms polymorphously regarding the type of certain registers.

In conclusion, the verification process which is the subject of the present invention, compared to the techniques of the prior art, makes it possible, on the one hand, to reduce the size of the program code which makes it possible to carry out the verification method, and on the other hand, to reduce the consumption of random-access memory during the verification operations, the degree of complexity being of the form $O(T_p+P_r)$ in the case of the verification process which is the subject of the present invention, instead of $(O(T_p+T_r)xN_b)$ for the verification process of the prior art, while however offering the same guarantees about the security of execution of the verified code.

Finally, the process of transforming original traditional code into standardized code is implemented by localized transformation of the code without transmitting additional information to the verifier component, i.e. the microprocessor card or on-board data-processing system.

Regarding the method of reallocating registers as described in figs. 4b and 5b, this method differs from the known methods of the prior art, as described in particular in US Patents 4,571,678 and 5,249,295, by the fact that:

- the register reallocation ensures that the same register cannot be assigned to two intervals with different main types, which thus guarantees that a given register is used with the same type throughout the code; and
- the existing register allocation algorithms, which are described in the abovementioned documents, assume a fixed number of registers, and attempt to minimize the transfers, called "*spills*", between registers and stack, whereas reallocation of registers as claimed in the subject of the present invention operates in a framework where the total number of registers is variable, as a consequence of which there is no purpose in carrying out transfers between registers and stacks when a process of minimizing the total number of registers is carried out.

The protocol for managing a program fragment downloaded onto an on-board system, and the methods of verifying this downloaded program fragment and of transforming this object code of a downloaded program fragment respectively, which are the subjects of the present invention, can of course be implemented in software.

Therefore, the present invention also concerns a computer program product which can be loaded directly into the internal memory of a reprogrammable on-board system, this on-board system making it possible to download a program fragment consisting of an object code, a series of instructions, executable by the microprocessor of the on-board system by way of a virtual machine equipped with an execution stack and with local registers or variables manipulated via these instructions so that this object code can be interpreted. The corresponding computer program product includes portions of object code to execute the protocol for managing a program fragment downloaded onto this on-board system, as shown in figs. 2 and 6 described above in

the description, when this on-board system is interconnected to a terminal and this program is executed by the microprocessor of this on-board system by way of the virtual machine.

The invention also concerns a computer program product which can be loaded directly into the internal memory of a reprogrammable on-board system, such as a microprocessor card with a rewritable memory, as shown with reference to fig. 6. This computer program product includes portions of object code to execute the stages of verifying a program fragment downloaded onto this on-board system, as shown and described above in the description, with reference to figs. 3a to 3j. This verification is executed when this on-board system is interconnected to a terminal and this program is executed by the microprocessor of this on-board system via the virtual machine.

The invention also concerns a computer program product; this computer program product includes portions of object code to execute the stages of the method of transforming the object code of a program fragment into standardized object code for this same program fragment, as shown in figs. 4a, 4b, 5a, 5b and 6, and described above in the description.

The present invention also concerns a computer program product which is recorded on a medium which can be used in a reprogrammable on-board system, e.g. a microprocessor equipped with a rewritable memory, this on-board system making it possible to download a program fragment consisting of an object code executable by this microprocessor, by way of a virtual machine equipped with an execution stack and local variables or registers manipulated via these instructions, to allow interpretation of this object code. The abovementioned computer program product includes, at least, a module of programs which can be read by the microprocessor of the on-board system via the virtual machine, to command execution of a procedure for managing the downloading

of a downloaded program fragment, as shown in fig. 2 and described above in the description, a module of programs which can be read by the microprocessor via the virtual machine, to command execution of a procedure for verifying, instruction by instruction, the object code which makes up this program fragment, as shown and described in relation to figs. 3a to 3j in the description above, and a module of programs which can be read by the microprocessor of this on-board system via the virtual machine, to command execution of a downloaded program fragment following or in the absence of a transformation of the object code of this program fragment into a standardized object code for this same program fragment, as shown in fig. 2.

The abovementioned computer program product also includes a module of programs which can be read by the microprocessor via the virtual machine, to command inhibition of execution, on the on-board system, of the program fragment in the case of an unsuccessful verification procedure of the abovementioned program fragment, as shown and described above in the description with reference to fig. 2.

ANNEXES

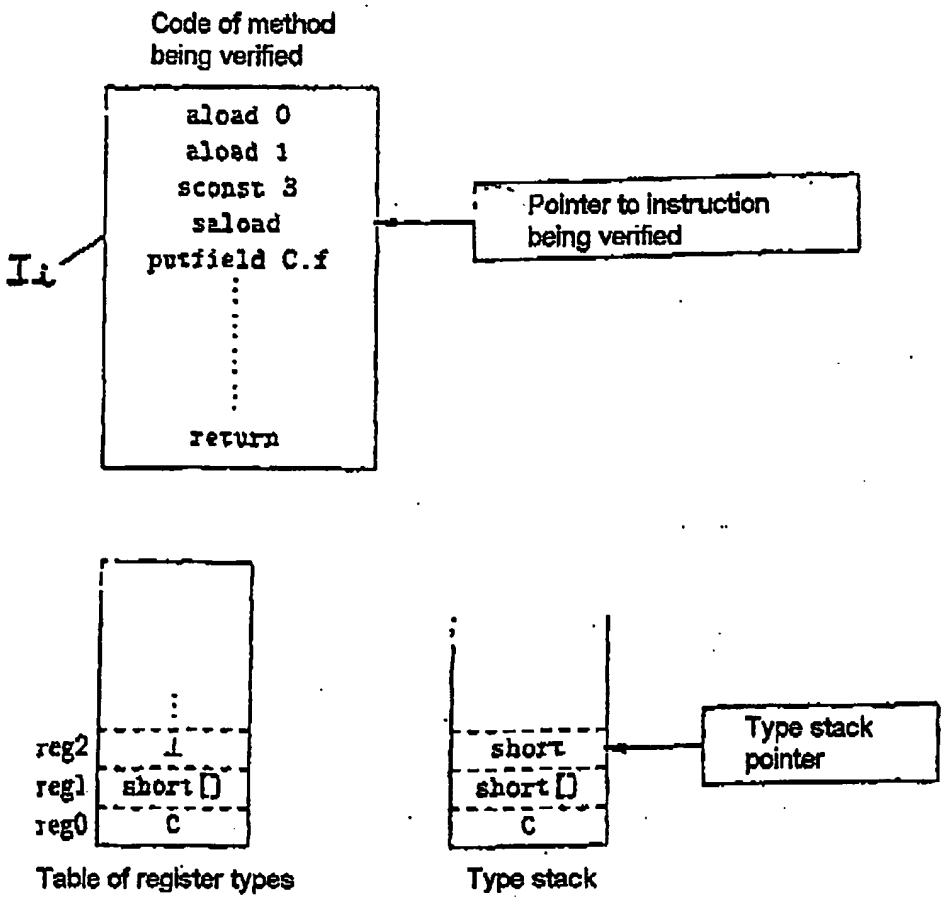

Code of method being verified
aload 0
aload 1
sconst 3
saload
putfield C.f
return
Ii
Pointer to instruction being verified
reg2
reg1
reg0
⊥
short []
C
Table of register types
short
short []
C
Type stack
Type stack pointer

TABLE 2

Pseudo-code of verifier module

PSEUDO-CODE OF VERIFIER MODULE

Global variables used:

T_r	number of registers declared by current method
T_p	maximum size of stack declared by current method
$tr[T_r]$	table of register types (402 in fig. 4)
$tp[T_p]$	stack type (403 in fig. 4)
pp	stack pointer (404 in fig. 4)
chg	flag indicating whether *tr* has changed

Initialize $pp \leftarrow 0$

Initialize $tp[0] \ldots tp[n-1]$ from types of n arguments of method

Initialize $tp[n] \ldots tp[T_r-1]$ to $\perp$

Initialize *chg* to true

While *chg* is true:

Reset *chg* to false

Position on first instruction of method

While end of method is not reached:

If current instruction is target of a branching instruction:

If $pp \neq 0$, verification fails

If current instruction is target of a subroutine call:

If previous instruction continues in sequence, failure

Take $tp[0] \leftarrow$ etaddr and $pp \leftarrow 1$

If current instruction is an exception handler of class *C*:

If previous instruction continues in sequence, failure

Do $tp[0] \leftarrow C$ and $pp \leftarrow 1$

If current instruction is a target of different kinds:

Verification fails

Determine types $a_1 \ldots a_n$ of arguments of instruction

If $pp < n$, failure (stack overflow)

For $i = 1, \ldots, n$:

If $tp[pp-n-i-1]$ is not subtype of a_i, failure

Do $pp \leftarrow pp-n$

Determine types $r_1, \ldots, r_m$ of results of instruction

If $pp+m \geq T_p$, failure (stack overflow)

For $i = 1, \ldots, m$, do $tp[pp+i-1] \; ? \; r_i$

Do $pp \leftarrow pp+m$

If current instruction is a write to a register *r*:

Determine type t of value written to record

Do $tr[r] \leftarrow lower\ bound\ (t, tr[r])$

If $tr[r]$ has changed, do $chg \leftarrow$ true

If current instruction is a branching:

If $pp \neq 0$, verification failure

Advance to next instruction

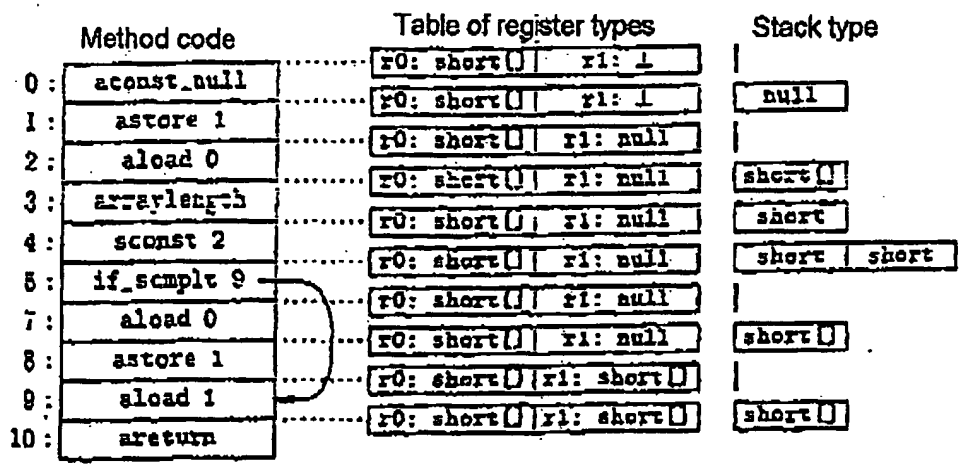
Return verification success code

TABLE T3

```
static short[] meth(short [] table    )
{
    short[] result      = null;
    if ( table length    >= 2) result = table;
     return table
}
```

TABLE T4

First iteration on method code:



Second iteration on method code:

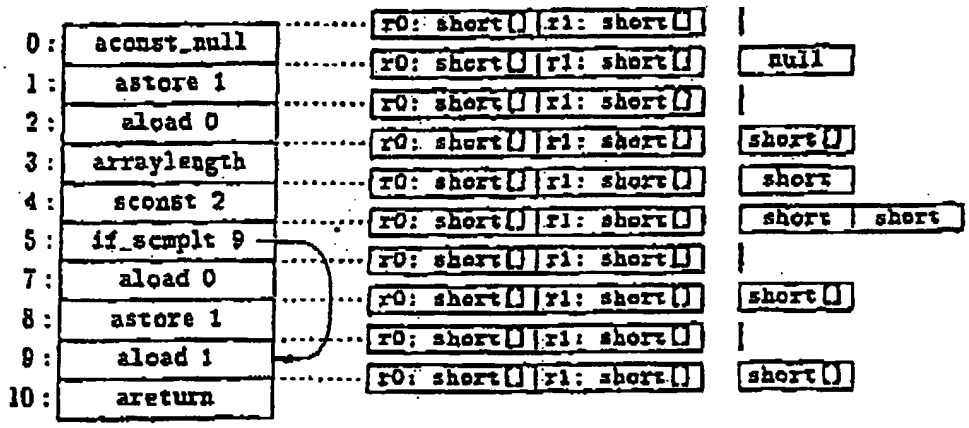
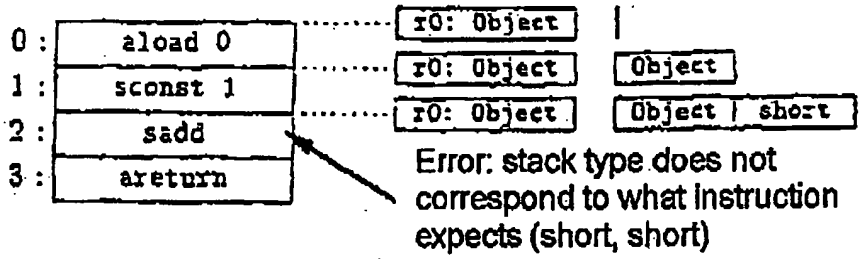
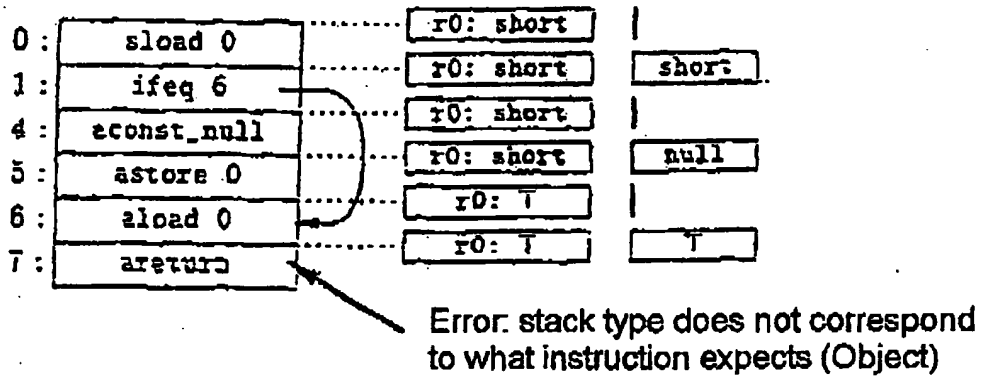


TABLE T5

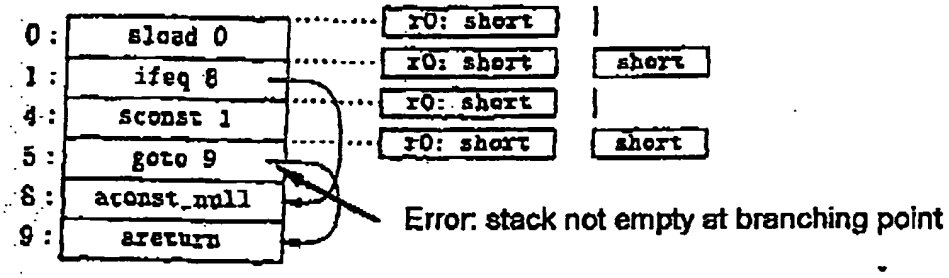
(a) *Violation of type constraints on arguments of an instruction:*



(b) *Inconsistent use of a register:*



(c) *Branchings introducing inconsistencies at stack level:*



(d) *Stack overflow within a loop:*

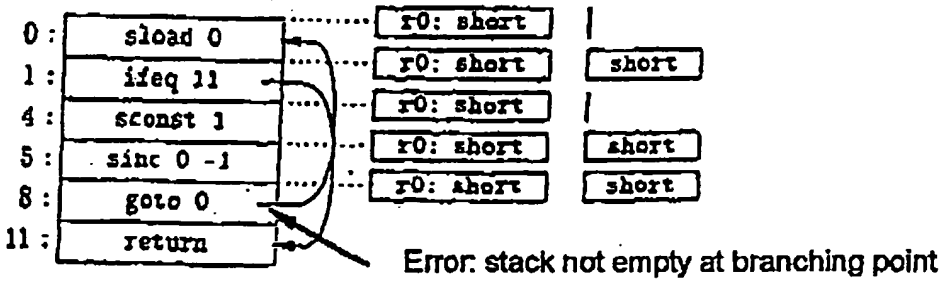


TABLE T6

(a) *Initial code of method, annotated by types of registers and of stack:*

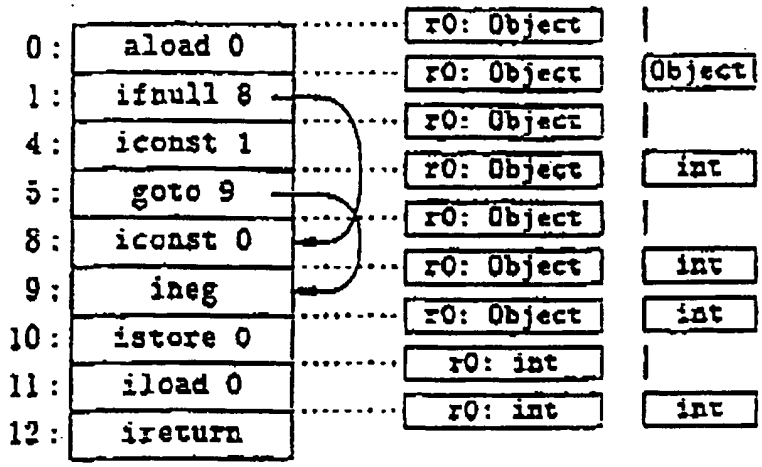


TABLE T7

(b) *Method code after standardization of stack at branching 5 → 9:*

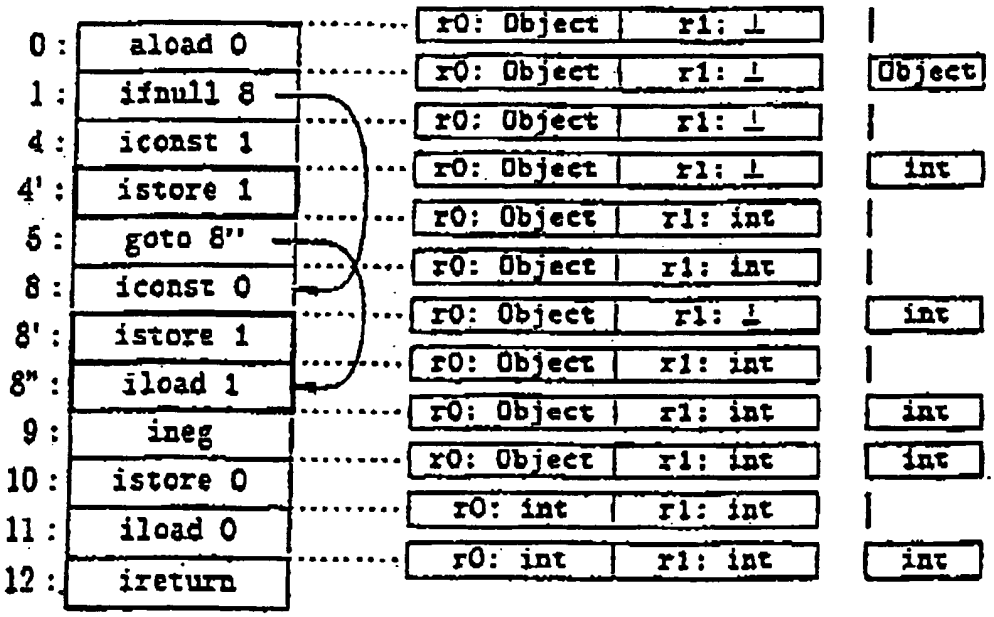


TABLE T8

(c) *Method code after reallocation of registers:*

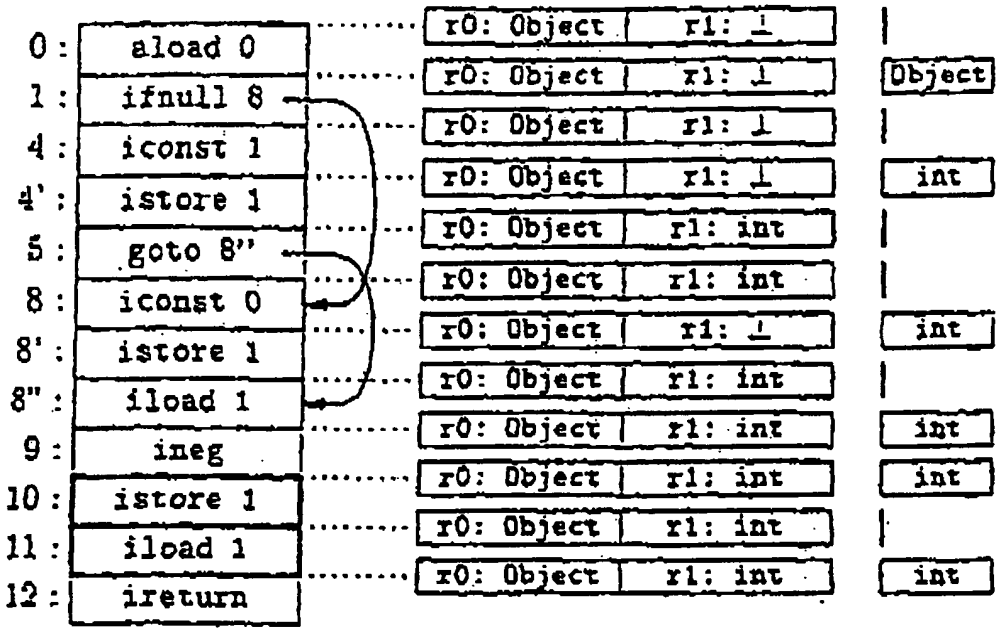


TABLE T9

(a) *Branching target, previous instruction not continuing in sequence:*

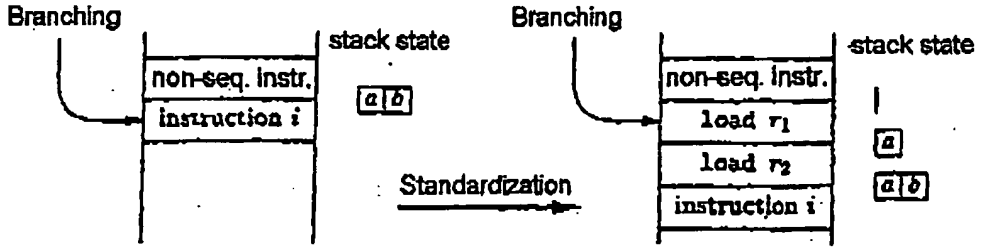


TABLE T10

(b) *Branching target, previous instruction continuing in sequence:*

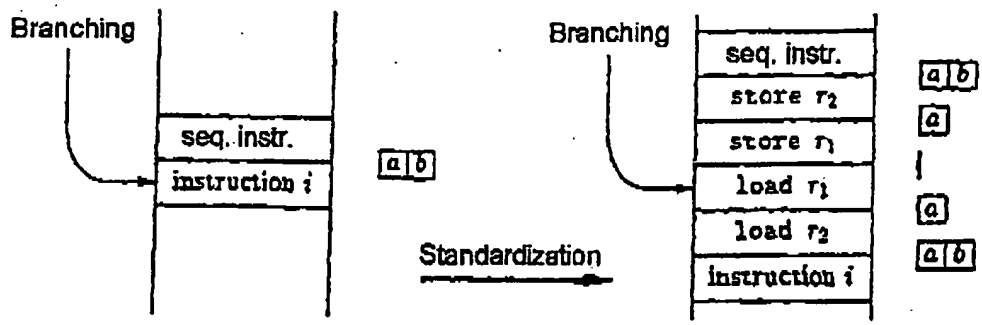


TABLE T11

(c) *Unconditional branching without arguments:*

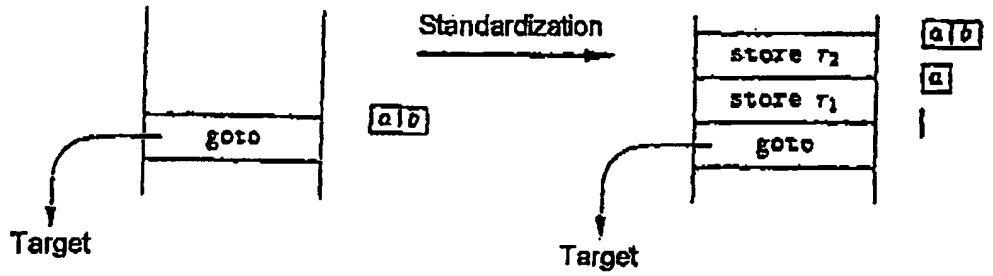
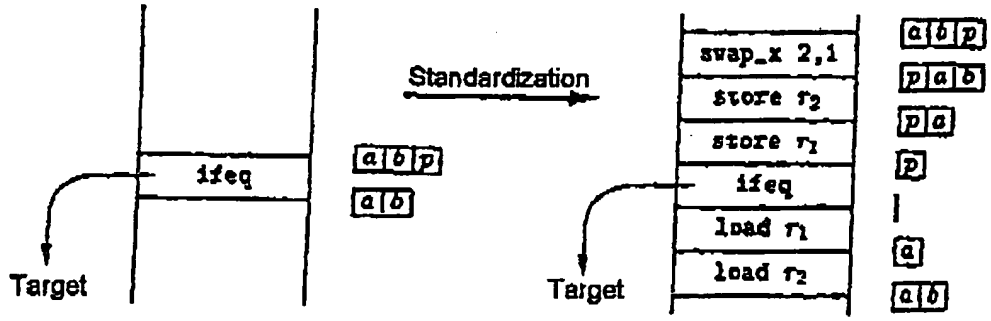


TABLE T12

(c) *Conditional branching with one argument:*



CLAIMS

1. A protocol for managing a program fragment downloaded onto a reprogrammable on-board system, such as a microprocessor card equipped with a rewritable memory, said program fragment consisting of an object code, a series of instructions, executable by the microprocessor of the on-board system by way of a virtual machine equipped with an execution stack and with local variables or registers manipulated via these instructions and making it possible to interpret this object code, said on-board system being interconnected to a terminal, characterized in that this protocol consists at least, at the level of said on-board system:

a) in detecting a command for downloading of this program fragment; and, on a positive response to this stage consisting in detecting a downloading command,

b) in reading the object code constituting this program fragment and in temporarily storing this object code;

c) in subjecting the whole of the object code stored temporarily in memory to a verification process, instruction by instruction, this verification process consisting at least in a stage of initializing the type stack and the table of register types representing the state of said virtual machine at the start of the execution of the temporarily stored object code and in a succession of stages of verification, instruction by instruction, by discerning the existence, for each current instruction, of a target, branching-instruction target, target of an exception-handler call, or target of a subroutine call, and in a verification and an updating of the effect of said current instruction on the type stack and on the table of register types, and, in the event of a successful verification of said object code,

d) in recording the downloaded program fragment in a directory of available program fragments, and, in the event of an unsuccessful verification of said object code,

e) in inhibiting execution, on said on-board system, of said program fragment.

2. The protocol as claimed in claim 1, characterized in that said stage e) of inhibiting the execution consists:

f) in deleting the momentarily recorded program fragment, when omitting to record the latter in said directory of available program fragments, and

g) in sending an error code to said reader.

3.The protocol as claimed in claim 1 or 2, characterized in that, on a negative response to said stage a) consisting in detecting a downloading command, this consists:

b') in detecting a command to select an available program fragment from a directory of program fragments; and, on a positive response to this stage, consisting in detecting a command to select an available program fragment;

c') in calling said selected available program fragment;

d') in executing said called available program fragment via the virtual machine, with no dynamic verification of variable types, access rights to the objects which are manipulated by the called available program fragment, or overflow of the execution stack when each instruction is executed, and, on a negative response to this stage consisting in detecting a command to select an available program fragment,

e') in proceeding to process the standard commands of the on-board system.

4. A method of verifying a program fragment downloaded onto a reprogrammable on-board system, such

as a microprocessor card equipped with a rewritable memory, said program fragment consisting of an object code and including at least one subprogram, a series of instructions, by the microprocessor of the on-board system by way of a virtual machine equipped with an execution stack and with operand registers manipulated by these instructions, and making it possible to interpret this object code, said on-board system being interconnected to a reader, characterized in that said method, following the detection of a downloading command and the storage of said object code constituting this program fragment in said rewritable memory, consists, for each subprogram:

α) in carrying out a stage of initializing the type stack and the table of register types by data representing the state of the virtual machine at the start of the execution of the temporarily stored object code;

β) in carrying out a verification of said temporarily stored object code instruction by instruction, by discerning the existence, for each current instruction, of a target, a branching-instruction target, a target of an exception-handler call or a target of a subroutine call;

γ) in carrying out a verification and an updating of the effect of said current instruction on the data types of said type stack and of said table of register types, on the basis of the existence of a branching-instruction target, of a target of a subroutine call or of a target of an exception-handler call, said verification being successful when the table of register types is not modified in the course of a verification of all the instructions, and the verification process being carried out instruction by instruction until the table of register types is stable, with

no modification present, the verification process being interrupted otherwise.

5. The verification method as claimed in claim 4, characterized in that the variable types which are manipulated during the verification process include at least:

- class identifiers corresponding to object classes which are defined in the program fragment;
- numeric variable types including at least a type short, an integer coded on p bits, and a type retaddr for the return address of a jump instruction JSR;
- a type null relating to references of null objects;
- a type object relating to objects;
- a first specific type $\perp$, representing the intersection of all the types and corresponding to the value 0, nil;
- a second specific type T, representing the union of all the types and corresponding to any type of value.

6. Method as claimed in claim 5, characterized in that all said variable types verify a subtyping relation:

$$\text{object} \varepsilon T;$$

$$\text{short, retaddr} \varepsilon T;$$

$$\perp \varepsilon \text{null, short, retaddr}.$$

7. The method as claimed in one of claims 4 to 6, characterized in that when said current instruction is the target of a branching instruction, said verification method consists in verifying that the type stack is empty, the verification process being continued for the following instruction in the case of a positive verification, and the verification process failing and the program fragment being rejected otherwise.

8. The method as claimed in one of claims 4 to 7, characterized in that when said current instruction is the target of a subroutine call, said verification process verifies that the previous instruction is an unconditional branching, a subroutine return or a raising of an exception, said verification process, in the case of a positive verification, proceeding to reupdate the stack of variable types by an entity of retaddr type, the return address of the subroutine, and the verification process failing and the program fragment being rejected otherwise.

9. The method as claimed in one of claims 4 to 8, characterized in that when the current instruction is the target of an exception handler, said verification process verifies that the previous instruction is an unconditional branching, a subroutine return or a raising of an exception, said verification process, in the case of a positive verification, proceeding to reupdate the type stack by entering the exception type, and the verification process failing and the program fragment being rejected otherwise.

10. The method as claimed in one of claims 4 to 9, characterized in that when the current instruction is the target of multiple incompatible branchings, the verification process fails and the program fragment is rejected.

11. The method as claimed in one of claims 4 to 10, characterized in that when the current instruction is not the target of any branching, the verification process continues by passing to an update of the type stack.

12. The method as claimed in one of claims 4 to 11, characterized in that the stage of verification of the effect of the current instruction on the type stack includes, at least:

- a stage of verifying that the type execution stack includes at least as many entries as the current instruction includes operands;

- a stage of unstacking and of verifying that the types of the entries at the top of the stack are subtypes of the types of the operands of the operands of this instruction;
- a stage of verifying the existence of a sufficient memory space on the type stack to proceed to stack the results of the current instruction;
- a stage of stacking on the stack data types which are assigned to these results.

13. The method as claimed in claim 12, characterized in that when the current instruction is an instruction to read a register of address n, the verification process consists:

- in verifying the data type of the result of this reading, by reading the entry n in the table of register types;
- in determining the effect of the current instruction on the type stack by unstacking the entries of the stack corresponding to the operands of this current instruction and by stacking the data type of this result.

14. The method as claimed in claim 12, characterized in that when the current instruction is an instruction to write to a register of address m, the verification process consists:

- in determining the effect of the current instruction on the type stack and the type t of the operand which is written in this register of address m;
- in replacing the type entry of the table of register types at address m by the type immediately above the previously stored type and above the type t of the operand which is written in this register of address m.

15. A method of transforming an object code of a program fragment, in which the operands of each instruction belong to the data types manipulated by this instruction, the execution stack does not exhibit any overflow phenomenon, for each branching instruction, the type of the stack variables at this branching is the same as at the targets of this branching, into a standardized object code for this same program fragment, in which the operands of each instruction belong to the data types manipulated by this instruction, the execution stack does not exhibit any overflow phenomenon, the execution stack is empty at each branching instruction and at each branching-target instruction, characterized in that this method consists, for all the instructions of said object code:

- in annotating each current instruction with the data type of the stack before and after execution of this instruction, the annotation data being calculated by means of an analysis of the data stream relating to this instruction;
- in detecting, within said instructions and within each current instruction, the existence of branchings, or respectively of branching-targets, for which said execution stack is not empty, the detection operation being carried out on the basis of the annotation data of the type of stack variables allocated to each current instruction, and in the presence of detection of a non-empty execution stack,
- in inserting instructions to transfer stack variables on either side of these branchings or of these branching targets, respectively in order to empty the contents of the execution stack into temporary registers before this branching and to reestablish the execution stack from said temporary registers after this branching, and in not inserting any transfer instruction otherwise, making it possible to obtain a standard-

ized object code for this same program fragment, in which the execution stack is empty at each branching instruction and at each branching-target instruction, in the absence of any modification to the execution of said program fragment.

16. A method of transforming an object code of a program fragment, in which the operands of each instruction belong to the data types manipulated by this instruction, and an operand of given type written into a register by an instruction of this object code is reread from this same register by another instruction of this object code with the same given data type, into a standardized object code for this same program fragment, in which the operands of each instruction belong to the data types manipulated by this instruction, the same data type being allocated to the same register throughout said standardized object code, characterized in that this method consists, for all the instructions of said object code:

- in annotating each current instruction with the data type of the registers before and after execution of this instruction, the annotation data being calculated by means of an analysis of the data stream relating to this instruction;
- in carrying out a reallocation of the registers, by detecting the original registers employed with different types, by dividing these original registers into separate standardized registers, one standardized register for each data type used, and reupdating the instructions which manipulate the operands which use said standardized registers.

17. The method as claimed in claim 15, characterized in that the stage consisting in detecting, within said instructions and within each current instruction, the existence of branchings, or respectively of branching targets, for which the execution stack is

not empty, consists, following detection of each corresponding instruction of rank i:

- in associating with each instruction of rank i a set of new registers, one new register being associated with each stack variable which is active at this instruction;
- in examining each detected instruction of rank i and in discerning the existence of a branching target or branching, respectively, and, in the case where the instruction of rank i is a branching target and that the execution stack at this instruction is not empty,
  - for every preceding instruction, of rank i-1, consisting of a branching, a raising of an exception or a program return, the detected instruction of rank i being accessible only by a branching,
  - •• in inserting a set of load instructions load to load from the set of new registers before said detected instruction of rank i, with redirection of all branchings to the detected instruction of rank i to the first inserted load instruction load; and
  - for every preceding instruction, of rank i-1, continuing in sequence, the detected instruction of rank i being accessible simultaneously by a branching and from the preceding instruction of rank i-1,
  - •• in inserting a set of backup instructions store to back up to the set of new registers before the detected instruction of rank i, and a set of load instructions load to load from this set of new registers, with redirection of all the branchings to the detected instruc-

tion of rank i to the first inserted load instruction load, and, in the case where said detected instruction of rank i is a branching to a given instruction,

- for every detected instruction of rank i consisting of an unconditional branching,
  - in inserting, before the detected instruction of rank i, multiple backup instructions store, a backup instruction being associated with each new register; and
- for every detected instruction of rank i consisting of a conditional branching, and for a number m > 0 of operands manipulated by this conditional branching instruction,
  - in inserting, before this detected instruction of rank i, a permutation instruction, swap-x, at the top of the execution stack of the m operands of the detected instruction of rank i and the n following values, this permutation operation making it possible to collect at the top of the execution stack the n values to be backed up in the set of new registers, and
  - in inserting, before the instruction of rank i, a set of backup instructions store to back up to the set of new registers, and
  - in inserting, after the detected instruction of rank i, a set of load instructions load to load from the set of new registers.

18. The method as claimed in claim 16, characterized in that the stage consisting in reallocating

registers by detecting the original registers employed with different types consists:

- in determining the lifetime intervals of each register;
- in determining the main data type of each lifetime interval, the main data type of a lifetime interval j for a register r being defined by the upper bound of the data types stored in this register r by the backup instructions store belonging to the lifetime interval j;
- in establishing an interference graph between the lifetime intervals, this interference graph consisting of a non-oriented graph of which each peak consists of a lifetime interval, and of which the arcs between two peaks j_1 and j_2 exist if a peak contains a backup instruction addressed to the register of the other peak or vice versa;
- in translating the uniqueness of a data type which is allocated to each register in the interference graph, by adding arcs between all pairs of peaks of the interference graph while two peaks of a pair of peaks do not have the same associated main data type;
- in carrying out an instantiation of the interference graph, by assigning to each lifetime interval a register number, in such a way that different register numbers are assigned to two adjacent life intervals in the interference graph.

19. An on-board system which can be reprogrammed by downloading program fragments, including at least one microprocessor, one random-access memory, one input/output module, one electrically reprogrammable non-volatile memory and one permanent memory, in which are installed a main program and a virtual machine which makes it possible to execute the main program and at least one program fragment using said microprocessor,

characterized in that said on-board system includes at least one program module to manage and verify a downloaded program fragment, said management and verification program module being installed in the permanent memory.

20. An on-board system which can be reprogrammed by downloading program fragments, including at least one microprocessor, one random-access memory, one input/output module, one electrically reprogrammable non-volatile memory and one permanent memory, in which are installed a main program and a virtual machine which makes it possible to execute the main program and at least one program fragment using said microprocessor, characterized in that said on-board system includes at least one program module to manage and verify a downloaded program fragment in accordance with the protocol for managing a downloaded program fragment as claimed in one of claims 1 to 3, said management and verification program module being installed in the permanent memory.

21. The on-board system as claimed in claim 20, characterized in that it includes at least one subprogram module to verify a downloaded program fragment, in accordance with the verification process as claimed in one of claims 4 to 14.

22. A method of transforming an object code of a program fragment, in which the operands of each instruction belong to the data types manipulated by this instruction, the execution stack does not exhibit any overflow phenomenon, for each branching instruction, the type of stack variables at this branching is the same as at the targets of this branching, and an operand of given type written to a register by an instruction of this object code is reread from this same register by another instruction of this object code with the same given data type, into a standardized object code for this same program fragment, in which the operands of each instruction belong to the data types ma-

nipulated by this instruction, the execution stack does not exhibit overflow phenomenon, the execution stack is empty at each branching instruction and at each branching-target instruction, the same data type being assigned to the same register throughout said standardized object code, characterized in that said conversion system includes, at least, installed in the working memory of a development computer or workstation, a program module to transform this object code into a standardized object code in accordance with the method as claimed in one of claims 15 to 18, making it possible to generate a standardized object code for said program fragment, satisfying the criteria for verifying this downloaded program fragment.

23. A computer program product which can be loaded directly into the internal memory of a reprogrammable on-board system, such as a microprocessor card equipped with a rewritable memory, this on-board system making it possible to download a program fragment consisting of an object code, a series of instructions, executable by the microprocessor of the on-board system by way of a virtual machine equipped with an execution stack and with local variables or registers manipulated via these instructions and making it possible to interpret this object code, this computer program product including portions of object code to execute the protocol for managing a downloaded program fragment on this on-board system as claimed in one of claims 1 to 3, when this on-board system is interconnected to a terminal and this program is executed by the microprocessor of this on-board system by way of said virtual machine.

24. A computer program product which can be loaded directly into the internal memory of a reprogrammable on-board system, such as a microprocessor card equipped with a rewritable memory, this on-board system making it possible to download a program fragment consisting of an object code, a series of instruc-

tions, executable by the microprocessor of the on-board system by way of a virtual machine equipped with an execution stack and with operand registers manipulated via these instructions and making it possible to interpret this object code, this computer program product including portions of object code to execute the stages of verifying a program fragment downloaded onto this on-board system as claimed in one of claims 4 to 14, when this on-board system is interconnected to a terminal and this program is executed by the microprocessor of this on-board system by way of said virtual machine.

25. A computer program product including portions of object code to execute stages of the method of transforming an object code of a downloaded program fragment into a standardized object code for this same program fragment as claimed in one of claims 15 to 18.

26. A computer program product which is recorded on a medium which can be used in a reprogrammable on-board system, such as a microprocessor card equipped with a rewritable memory, this on-board system making it possible to download a program fragment consisting of an object code, a series of instructions, executable by the microprocessor of the on-board system by way of a virtual machine equipped with an execution stack and with local variables or registers manipulated via these instructions and making it possible to interpret this object code, this computer program product including, at least:

- program resources which can be read by the microprocessor of this on-board system via said virtual machine, to command execution of a procedure for managing the downloading of a downloaded program fragment;
- program resources which can be read by the microprocessor of this on-board system via said virtual machine, to command execution of a procedure for verifying, instruction by instruc-

tion, the object code which makes up said program fragment;

- program resources which can be read by the microprocessor of this on-board system via said virtual machine, to command execution of a downloaded program fragment following or in the absence of a conversion of the object code of this program fragment into a standardized object code for this same program fragment.

27. The computer program product as claimed in claim 26, additionally including program resources which can be read by the microprocessor of this on-board system via said virtual machine, to command inhibition of execution, on said on-board system, of said program fragment in the case of an unsuccessful verification procedure of this program fragment.

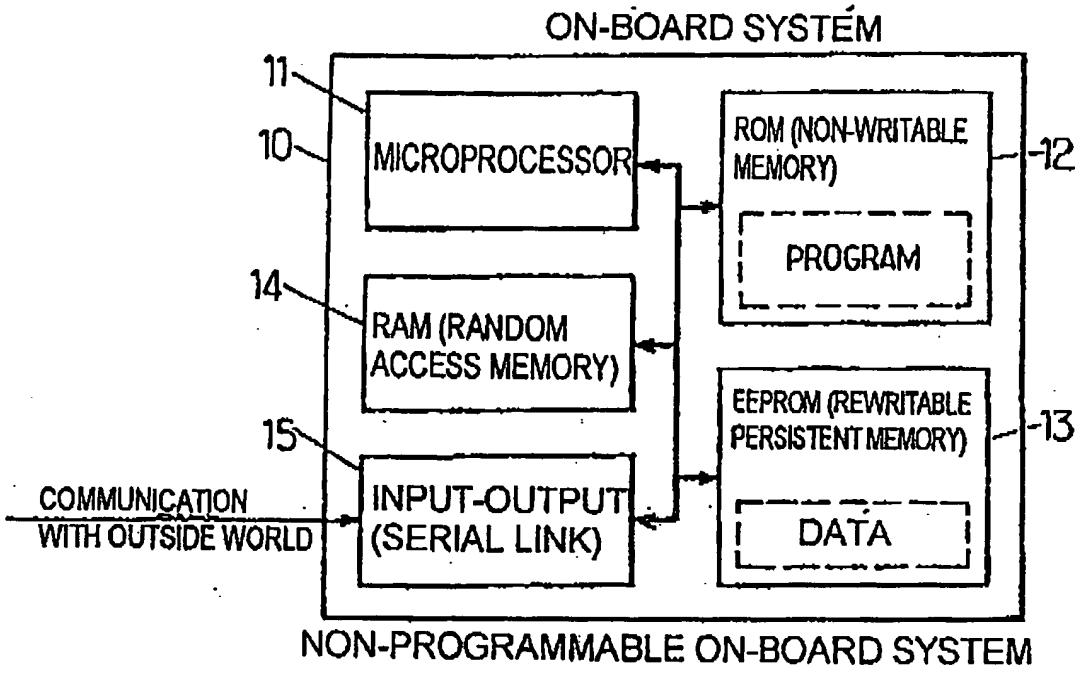


FIG.1a. (PRIOR ART)

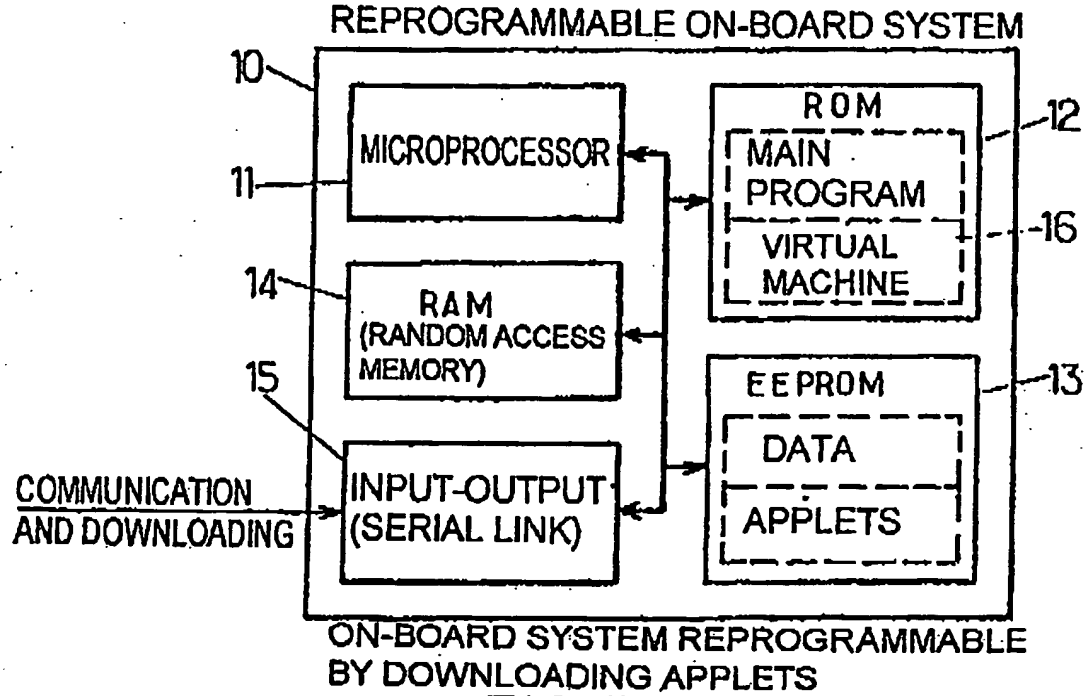


FIG.1b. (PRIOR ART)

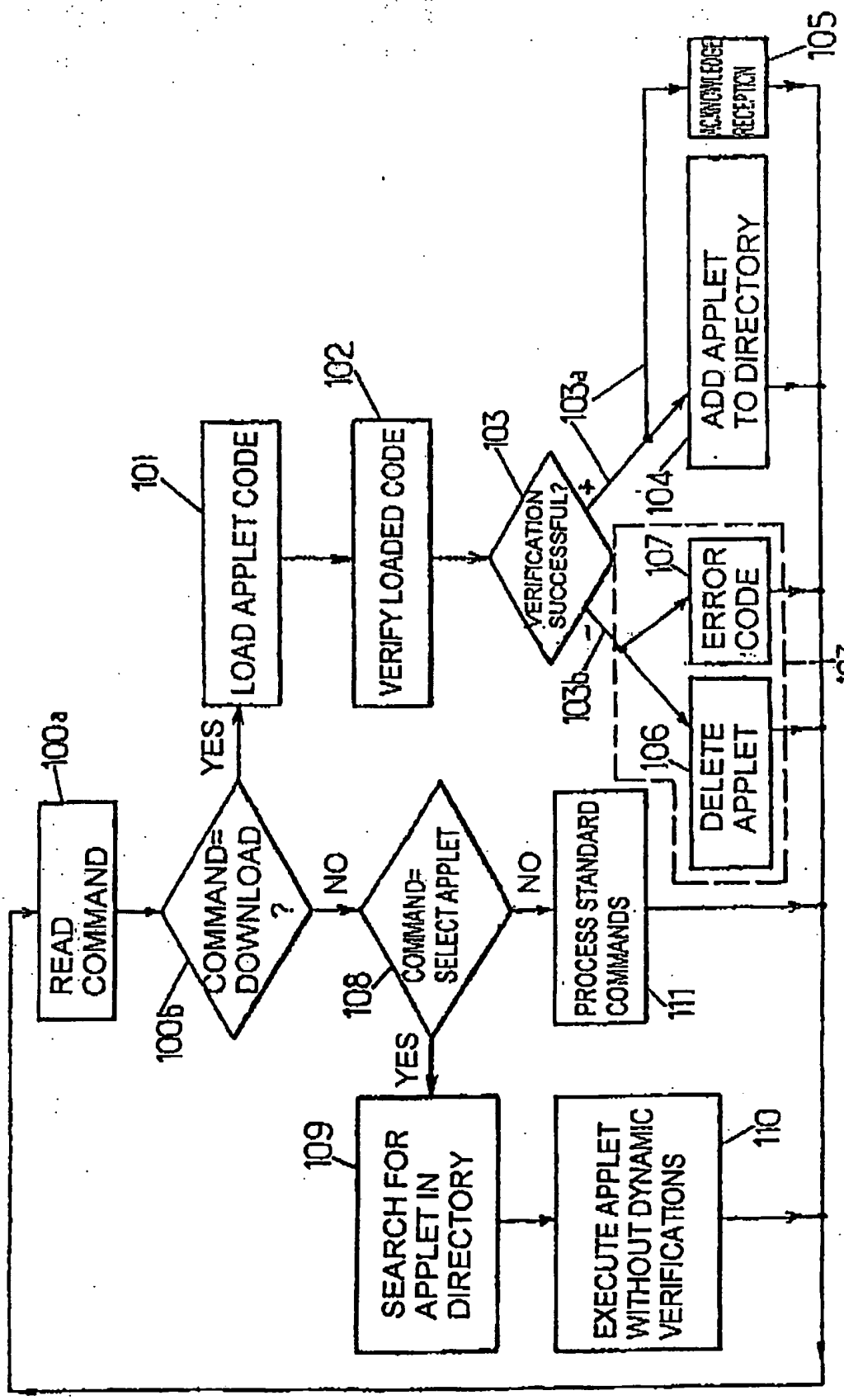


FIG.2. MANAGEMENT PROTOCOL

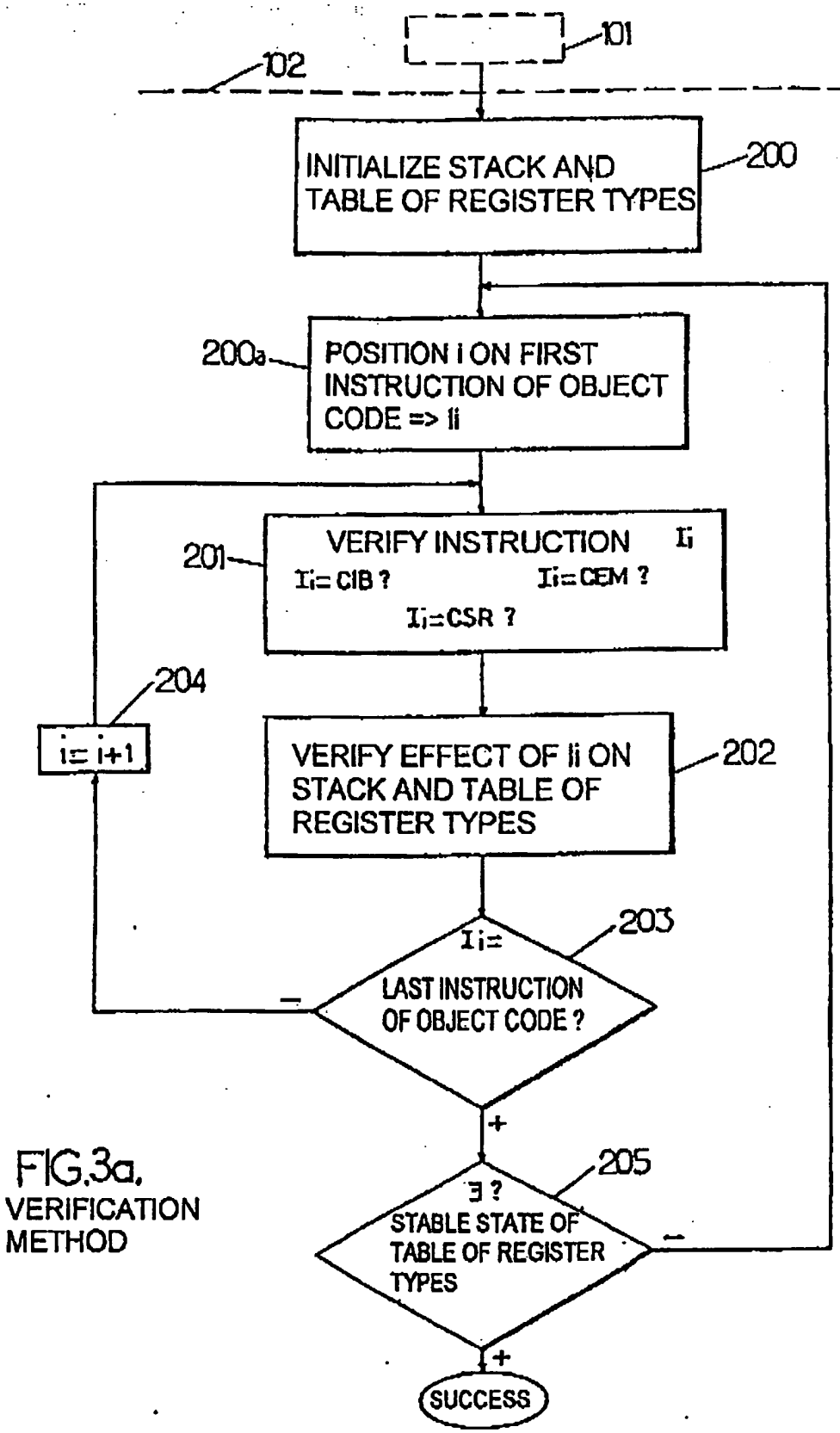


FIG.3a.
VERIFICATION METHOD

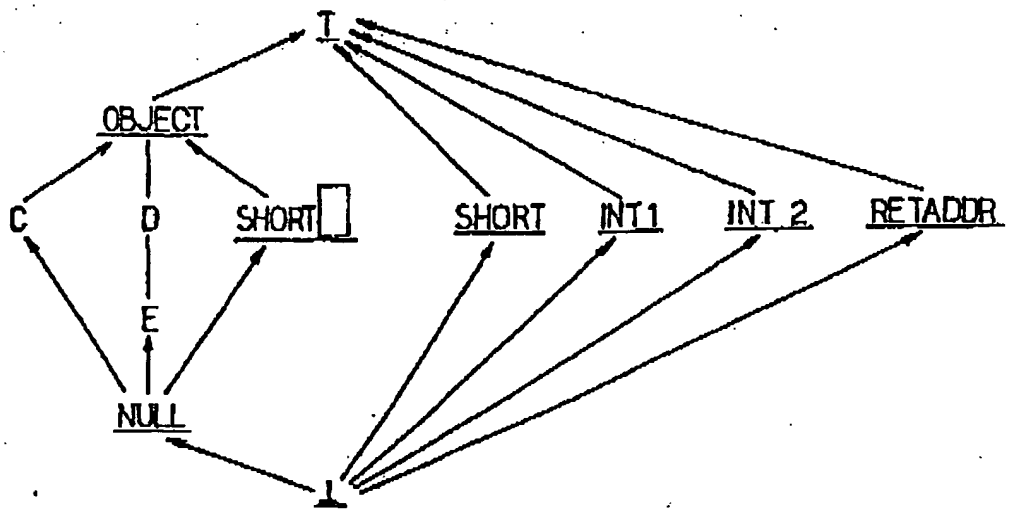


FIG.3b. DATA TYPES AND SUBTYPING RELATION

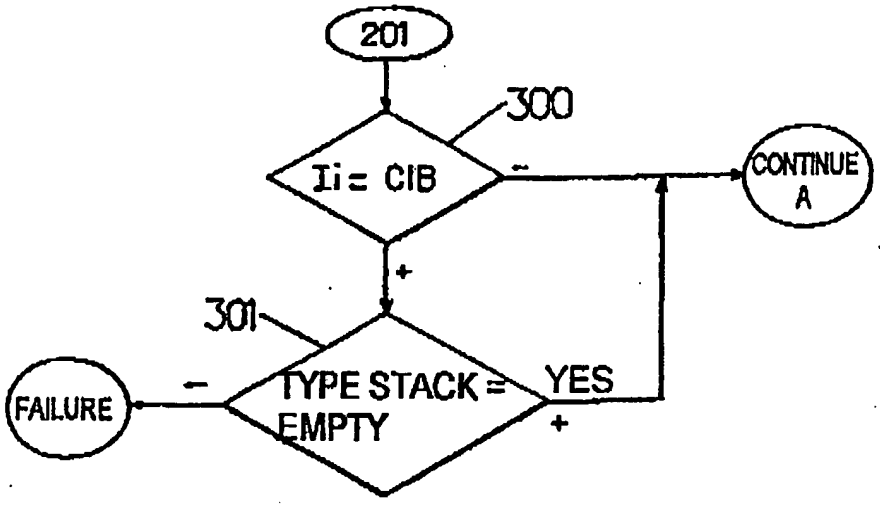


FIG.3c. VERIFICATION: MANAGEMENT OF BRANCHING INSTRUCTION

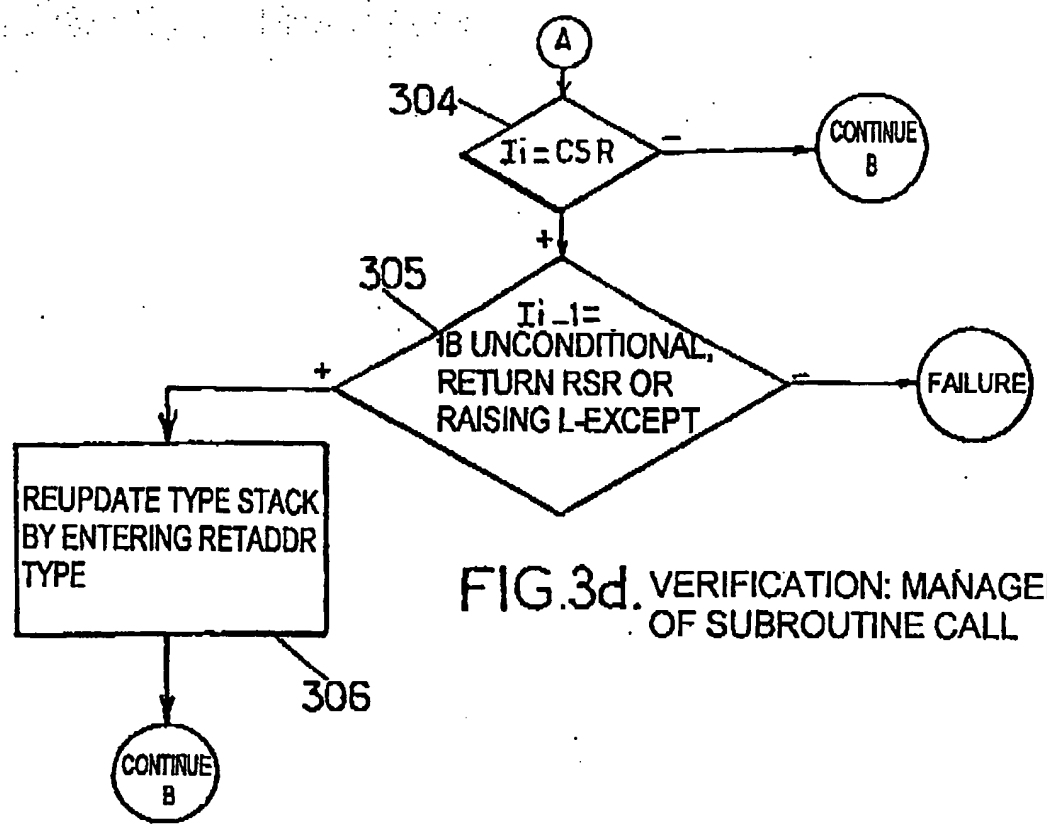


FIG.3d. VERIFICATION: MANAGEMENT OF SUBROUTINE CALL

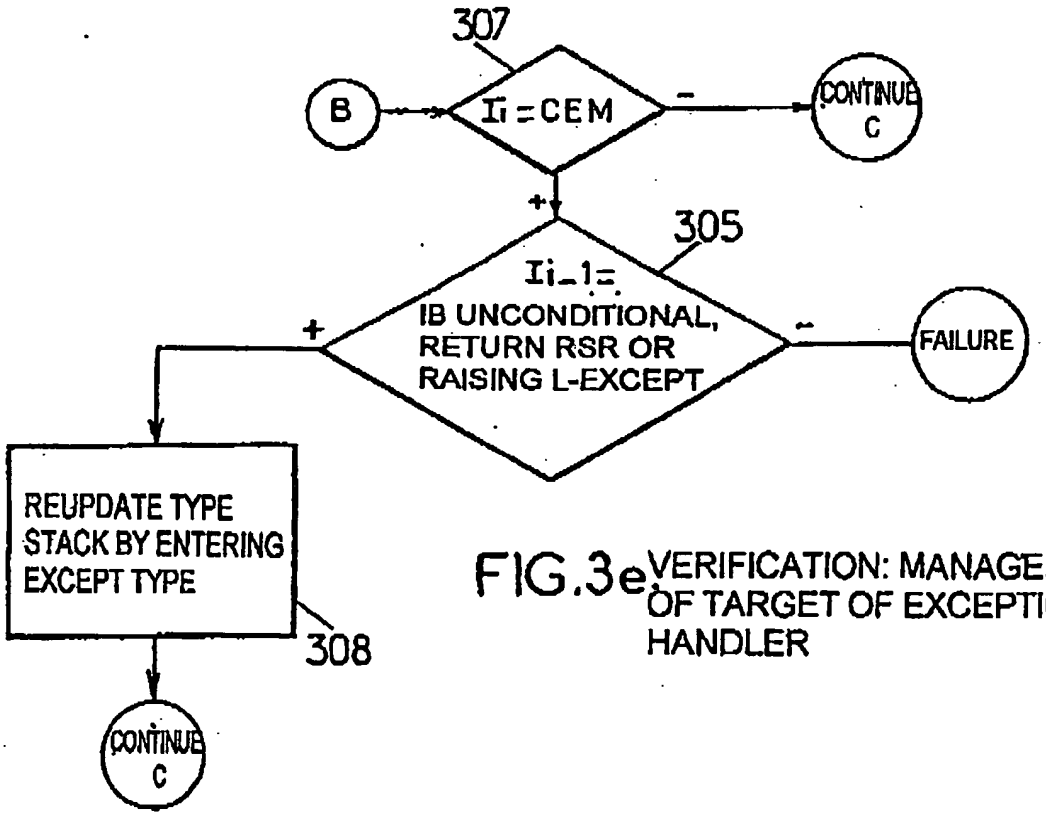


FIG.3e VERIFICATION: MANAGEMENT OF TARGET OF EXCEPTION HANDLER

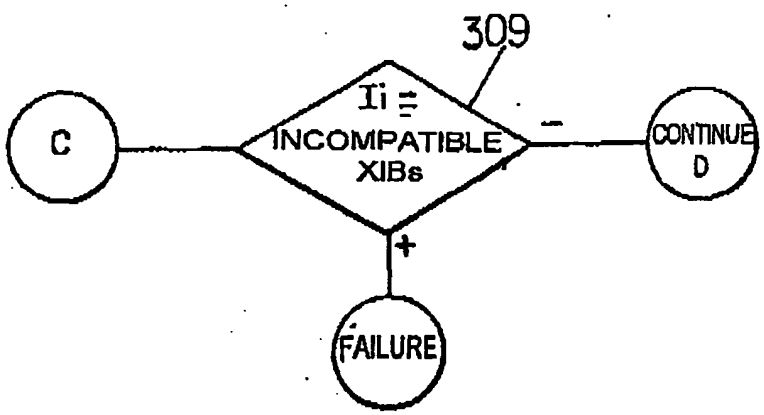


FIG.3f. VERIFICATION: MANAGEMENT OF TARGET OF INCOMPATIBLE BRANCHINGS

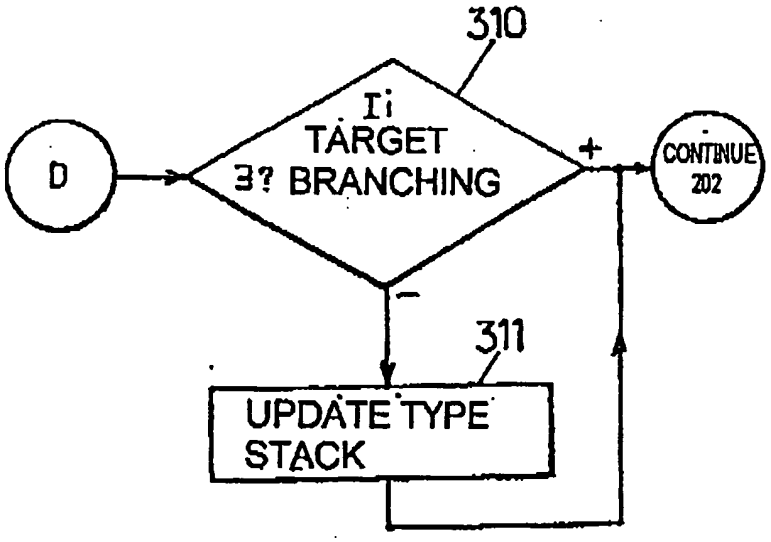


FIG.3g. VERIFICATION: MANAGEMENT OF ABSENCE OF BRANCHING TARGET

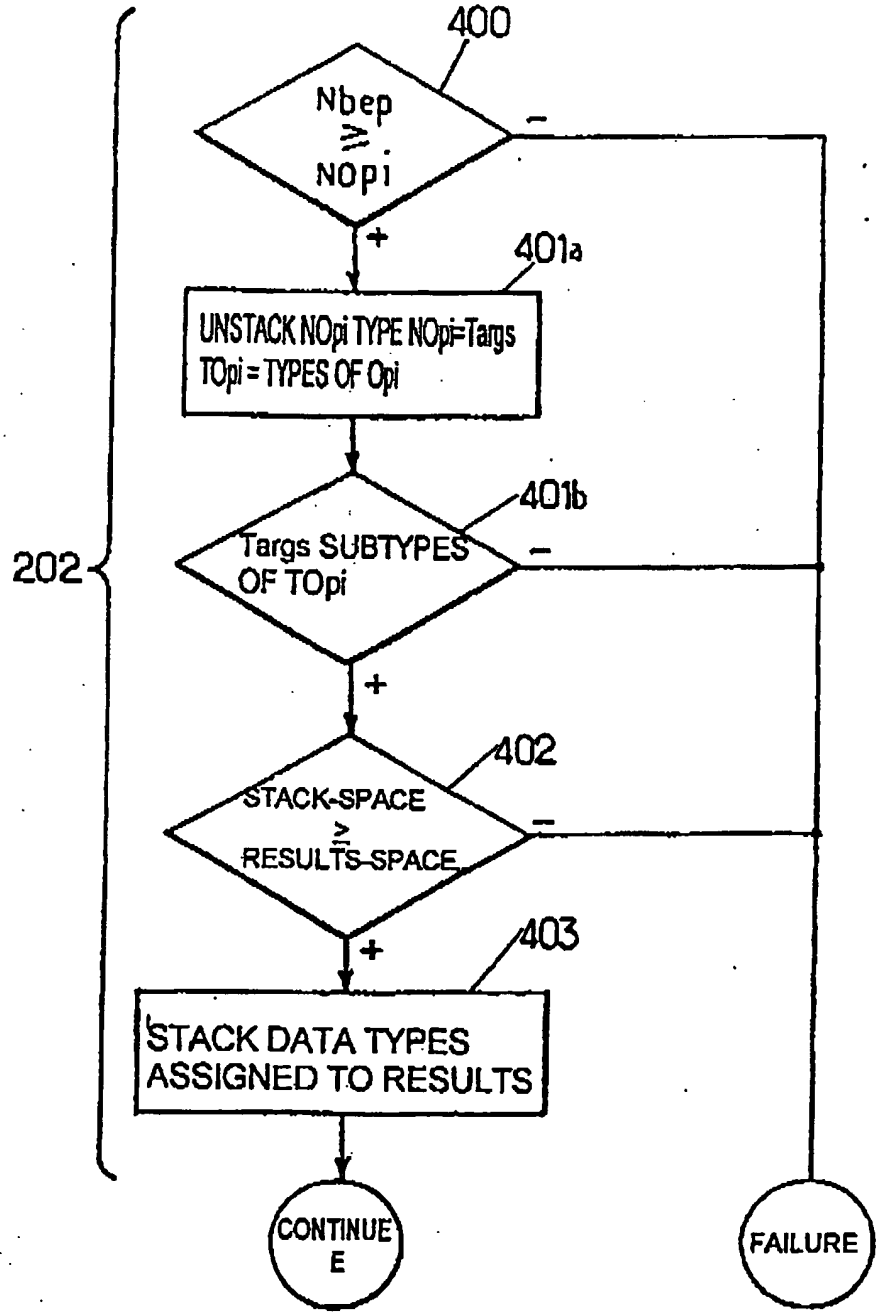


FIG.3h. VERIFICATION: EFFECT OF CURRENT INSTRUCTION ON TYPE STACK

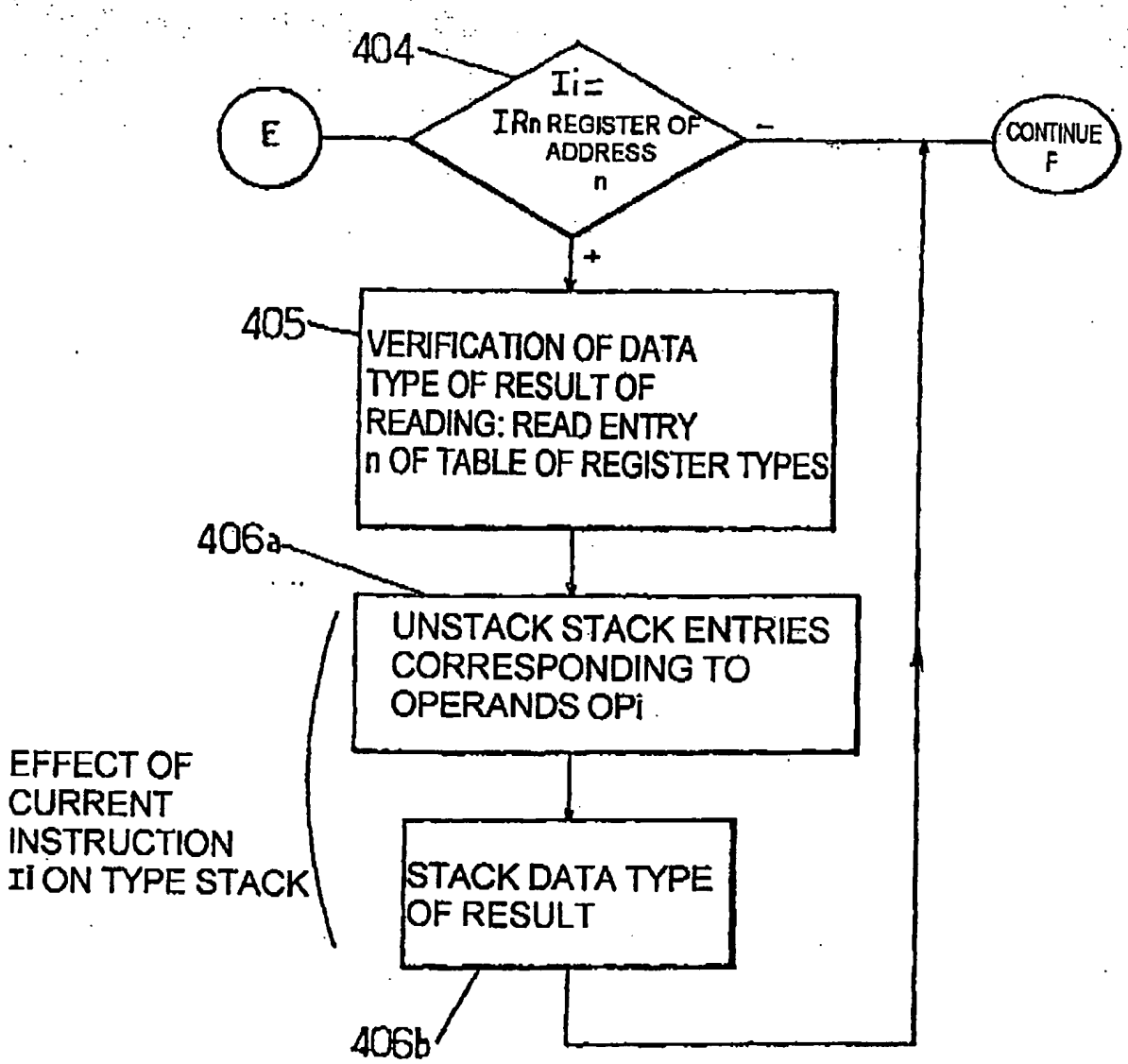


FIG.3i. VERIFICATION: MANAGEMENT OF INSTRUCTION TO READ REGISTER

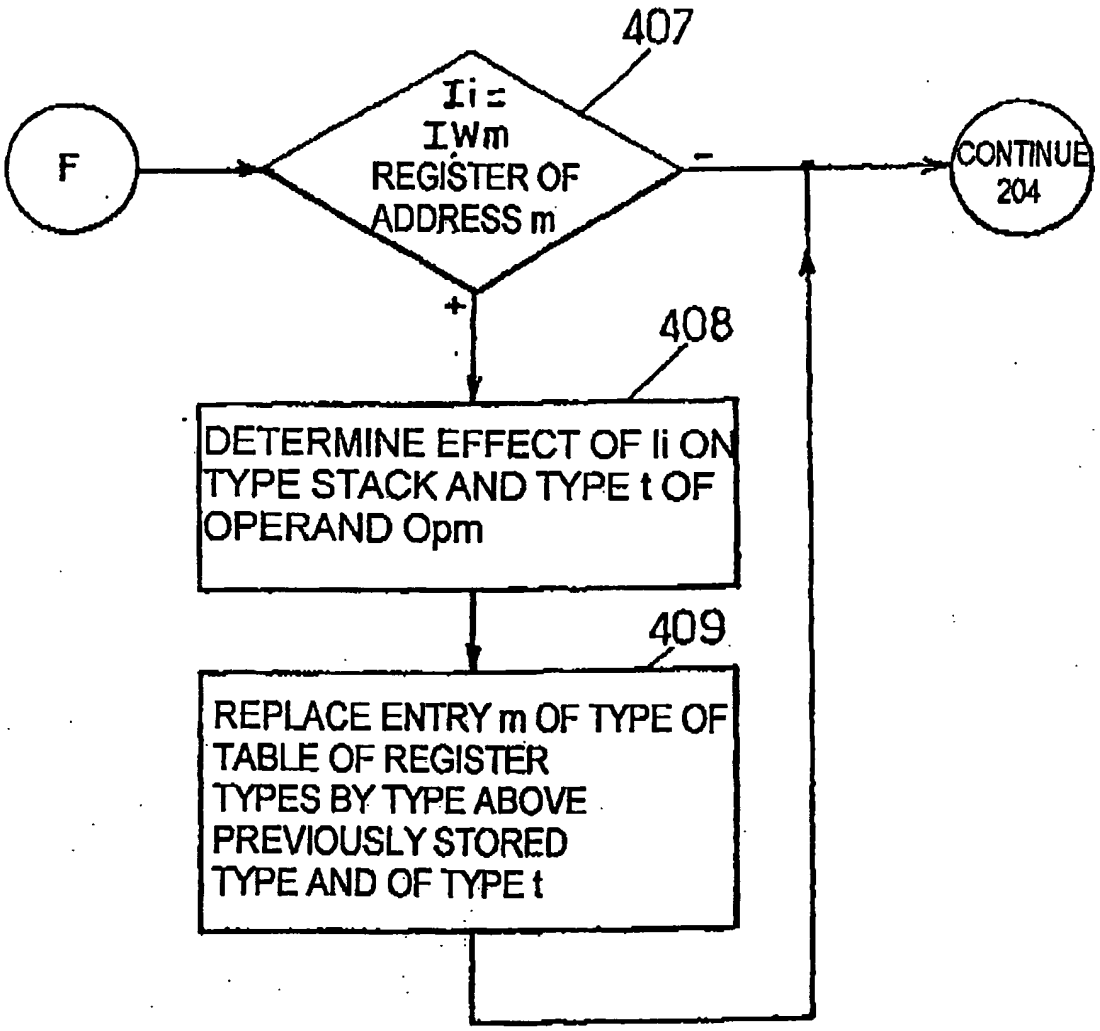


FIG.3j. VERIFICATION: MANAGEMENT OF INSTRUCTION TO WRITE REGISTER

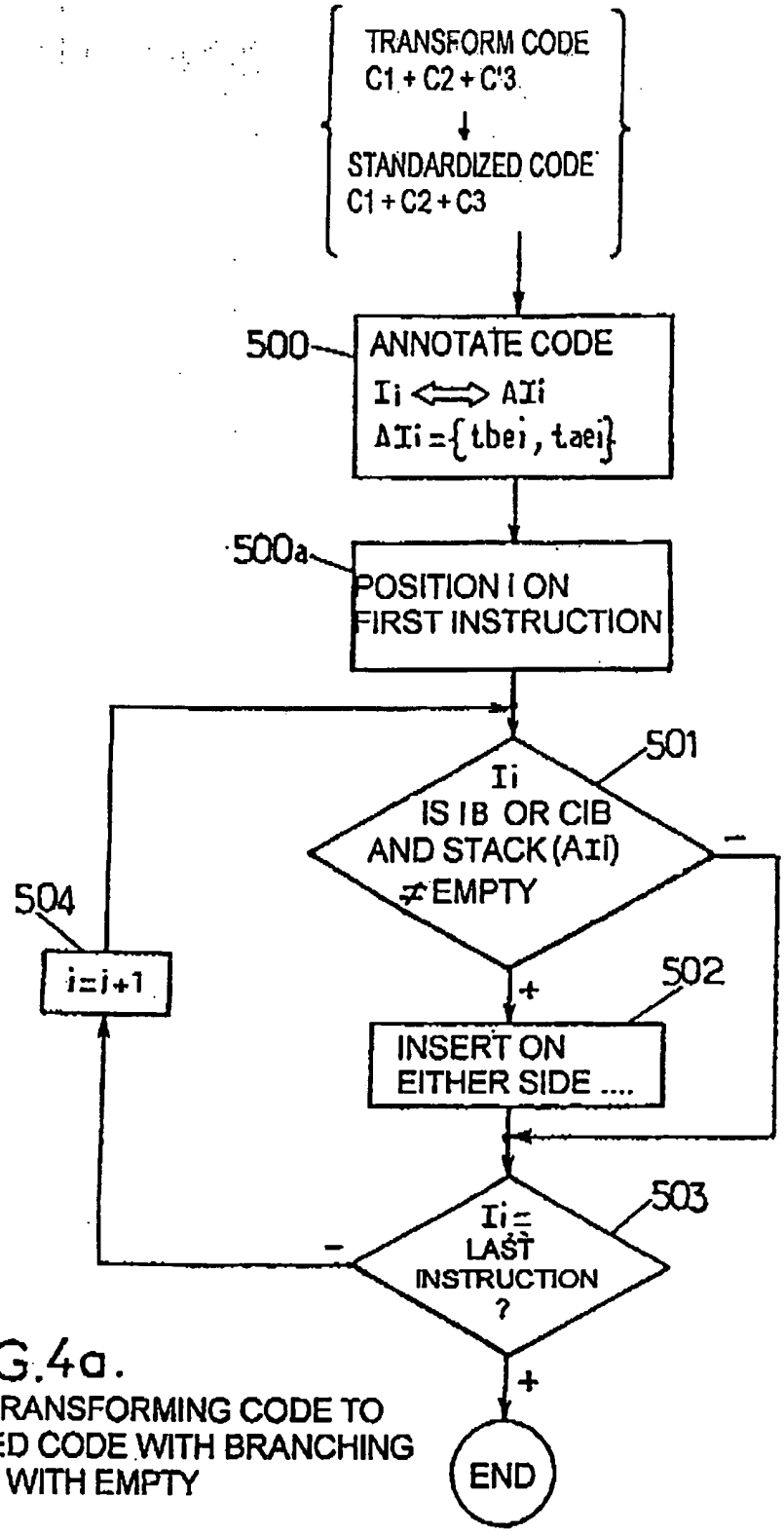


FIG.4a.

METHOD OF TRANSFORMING CODE TO STANDARDIZED CODE WITH BRANCHING INSTRUCTION WITH EMPTY STACK

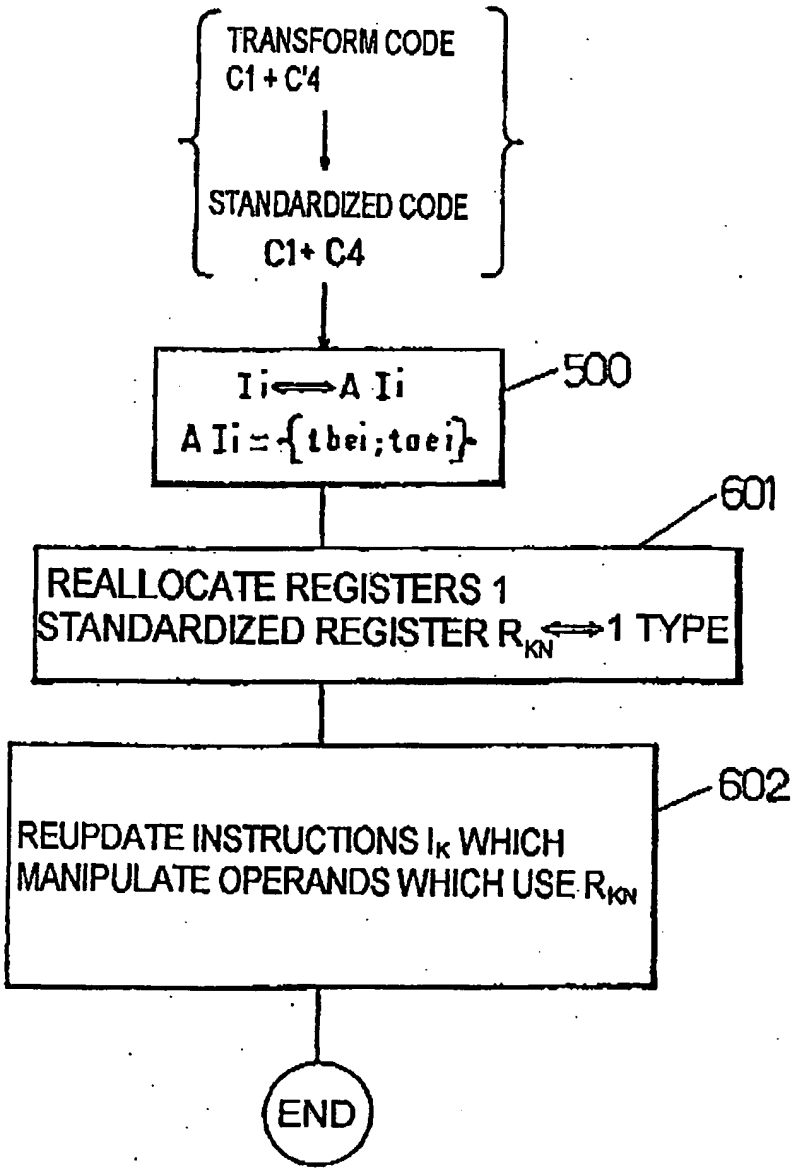


FIG.4b. METHOD OF TRANSFORMING CODE TO STANDARDIZED CODE USING TYPE REGISTERS

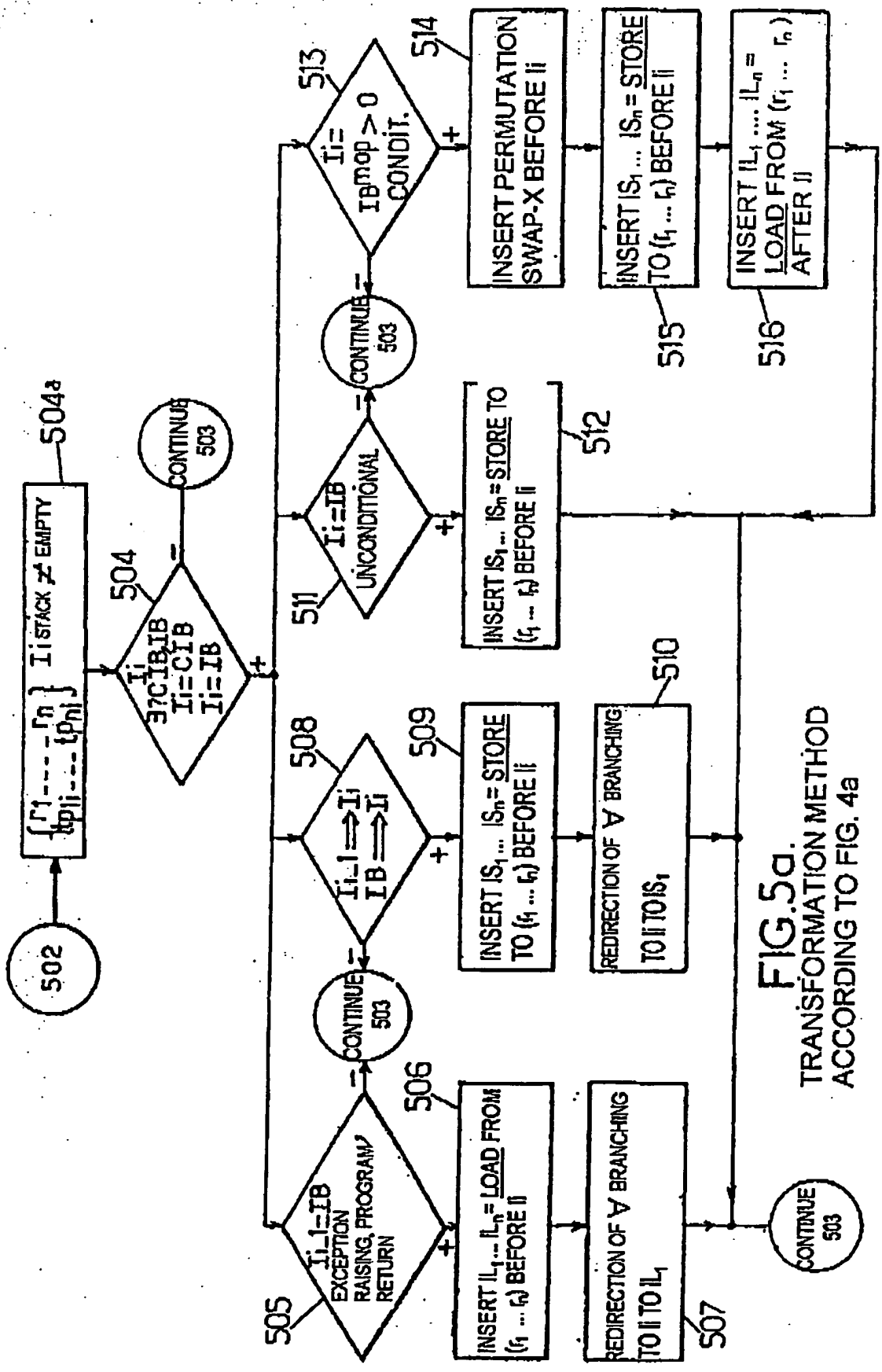


FIG.5a.
TRANSFORMATION METHOD
ACCORDING TO FIG. 4a

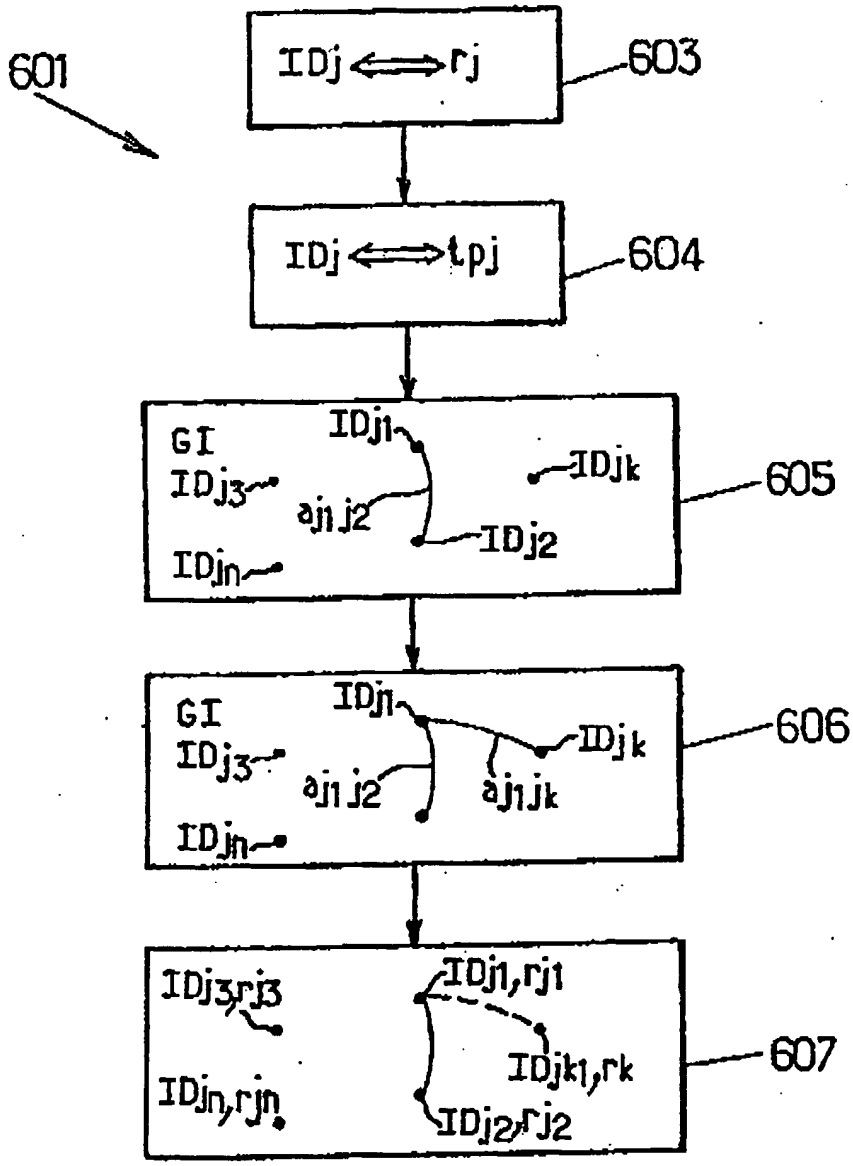


FIG.5b.

TRANSFORMATION METHOD ACCORDING TO FIG. 4b

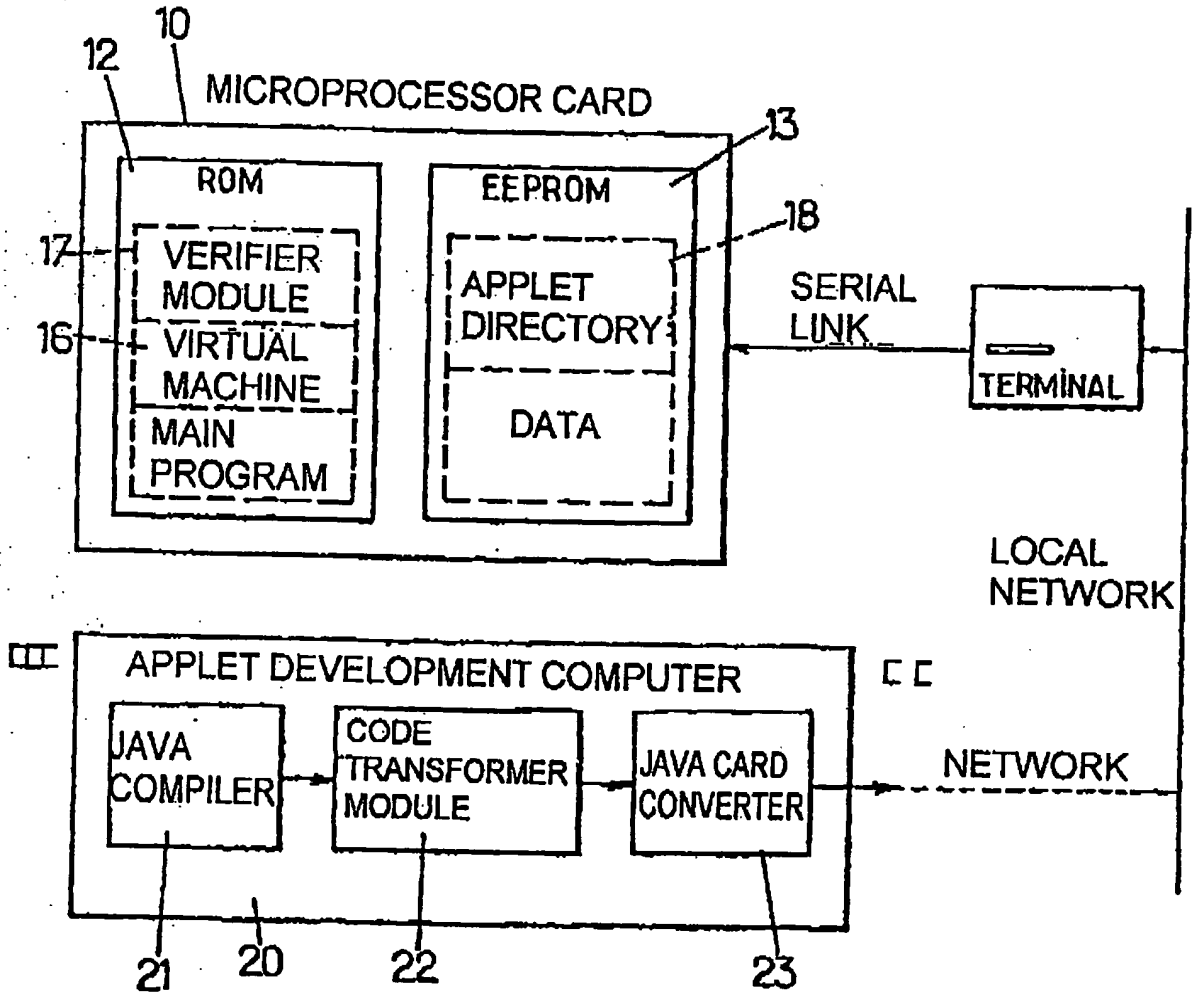

10
12
MICROPROCESSOR CARD
13
ROM
EEPROM
18
17
VERIFIER MODULE
APPLET DIRECTORY
16
VIRTUAL MACHINE
SERIAL LINK
MAIN PROGRAM
DATA
TERMINAL
LOCAL NETWORK
APPLET DEVELOPMENT COMPUTER
JAVA COMPILER
CODE TRANSFORMER MODULE
JAVA CARD CONVERTER
NETWORK
21
20
22
23

FIG.6.

This Page is Inserted by IFW Indexing and Scanning Operations and is not part of the Official Record

BEST AVAILABLE IMAGES

Defective images within this document are accurate representations of the original documents submitted by the applicant.

Defects in the images include but are not limited to the items checked:

- ☐ BLACK BORDERS
- ☐ IMAGE CUT OFF AT TOP, BOTTOM OR SIDES
- ☐ FADED TEXT OR DRAWING
- ☐ BLURRED OR ILLEGIBLE TEXT OR DRAWING
- ☐ SKEWED/SLANTED IMAGES
- ☐ COLOR OR BLACK AND WHITE PHOTOGRAPHS
- ☐ GRAY SCALE DOCUMENTS
- ☐ LINES OR MARKS ON ORIGINAL DOCUMENT
- ☐ REFERENCE(S) OR EXHIBIT(S) SUBMITTED ARE POOR QUALITY
- ☐ OTHER: _____

IMAGES ARE BEST AVAILABLE COPY.
As rescanning these documents will not correct the image problems checked, please do not report these problems to the IFW Image Problem Mailbox.